

COMPUTING
SCIENCE

Modelling and Analysis of Dynamic Reconfiguration in BP-calculus

F. Abouzaid, J. Mullins, M. Mazzara, N. Dragoni

TECHNICAL REPORT SERIES

No. CS-TR-1322

April 2012

Modelling and Analysis of Dynamic Reconfiguration in BP-calculus

F. Abouzaid, J. Mullins, M. Mazzara, N. Dragoni

Abstract

The BP-calculus is a formalism based on the pi-calculus and encoded in WS-BPEL. The BP-calculus is intended to specifically model and verify Service Oriented applications. One important feature of SOA is the ability to compose services that may dynamically evolve along runtime. Dynamic reconfiguration of services increases their availability, but puts accordingly heavy demands for validation, verification, and evaluation. In this paper we formally model and analyse dynamic reconfigurations and their requirements in BP-calculus and show how reconfigurable components can be modelled using handlers that are essential parts of WS-BPEL language.

Bibliographical details

ABOUZAID, F., MULLINS, J., MAZZARA, M., DRAGONI, N.

Modelling and Analysis of Dynamic Reconfiguration in BP-calculus

[By] F. Abouzaid, J. Mullins, M. Mazzara, N. Dragoni

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1322)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1322

Abstract

The BP-calculus is a formalism based on the pi-calculus and encoded in WS-BPEL. The BP-calculus is intended to specifically model and verify Service Oriented applications. One important feature of SOA is the ability to compose services that may dynamically evolve along runtime. Dynamic reconfiguration of services increases their availability, but puts accordingly heavy demands for validation, verification, and evaluation. In this paper we formally model and analyse dynamic reconfigurations and their requirements in BP-calculus and show how reconfigurable components can be modelled using handlers that are essential parts of WS-BPEL language.

About the authors

Faisal Abouzaid received a 3rd cycle Doctorate from University of Bordeaux (France) and a Ph.D degree from Polytechnic School of Montreal (Canada). From 1990 to 2002 he held a Professor of Computing Science position at University of Casablanca (Morocco) where he also served as the Director of the Computer Science Department. In 2004 he joined the CRAC laboratory of the Polytechnic School of Montreal as a Research Associate. His current research interests are: web services, orchestrations, dependability and formal methods.

John Mullins is Professor in the Department of Computer Engineering (DGI) at the Ecole Polytechnique de Montreal. He is also associate member of the LaCIM (Combinatorics and Mathematical Computer Sciences Laboratory), a research centre associated with the Mathematics Department, Computer Sciences Department and the Canada Research Chair in Algebra, Combinatorics and Mathematical Computer Sciences of the Universite du Quebec Montreal. John Mullins' research expertise includes applications of mathematical logic to analysis of concurrency and security: Models and calculi for concurrent systems (including object-oriented and mobile systems) and security systems (e.g. security controllers, security protocols), modal and temporal logics with fixed points and applications to system verification and description. He has published more than eighty papers in international journals and conferences in these fields. He has also introduced and successfully used these innovating methods in industries and government agencies (Defense Research and Development Canada, Bombardier Aerospace, CMC Electronics). In 2000, he has founded and currently heads the CRAC (Design and Realization of Complex Systems) laboratory and has succeeded with his team to detect many flaws in cryptographic protocols. Also, he has been or is currently PC member of several international conferences.

Manuel Mazzara achieved his Masters in 2002 and his Ph.D in 2006 at the University of Bologna. His thesis was based on Formal Methods for Web Services Composition. During 2000 he was a Technical Assistant at Computer Science Laboratories (Bologna, Italy). In 2003 he worked as Software Engineer at Microsoft (Redmond, USA). In 2004 and 2005 he worked as a freelance consultant and teacher in Italy. In 2006 he was an assistant professor at the University of Bolzano (Italy) and in 2007 a researcher and project manager at the Technical University of Vienna (Austria). Currently he is a Research Associate at the Newcastle University (UK) working on the DEPLOY project.

Nicola Dragoni obtained a M.Sc. Degree and a Ph.D. in computer science, respectively in 2002 and 2006, both at University of Bologna, Italy. He visited the Knowledge Media Institute at the Open University (UK) in 2004 and the MIT Center for Collective Intelligence (USA) in 2006. In 2007 and 2008 he was post-doctoral research fellow at University of Trento, working on the S3MS project. Between 2005 and 2008 he also worked as freelance IT consultant. In 2009 he joined Technical University of Denmark (DTU) as assistant professor in security and distributed systems. Since May 2011 he has been associate professor at the same university.

Suggested keywords

DYNAMIC RECONFIGURATION

PROCESS ALGEBRA

WS-BPEL

Modelling and Analysis of Dynamic Reconfiguration in BP-calculus.

F. Abouzaid¹, J. Mullins¹, M. Mazzara², and N. Dragoni³

¹ École Polytechnique de Montréal, Campus of the University of Montreal, Canada
{m.abouzaid, john.mullins}@polymtl.ca

² Newcastle University, Newcastle upon Tyne, UK and UNU-IIST, Macao
Manuel.Mazzara@ncl.ac.uk

³ Technical University of Denmark (DTU), Copenhagen
ndra@imm.dtu.dk

Abstract The BP-calculus is a formalism based on the π -calculus and encoded in WS-BPEL. The BP-calculus is intended to specifically model and verify Service Oriented Applications. One important feature of SOA is the ability to compose services that may dynamically evolve along runtime. Dynamic reconfiguration of services increases their availability, but puts accordingly, heavy demands for validation, verification, and evaluation. In this paper we formally model and analyze dynamic reconfigurations and their requirements in BP-calculus and show how reconfigurable components can be modeled using handlers that are essential parts of WS-BPEL language.

1 Introduction

In service-oriented computing (SOC), the correct composition of basic services is a key task and remains unsolved. The problem with industry-proposed languages for orchestration of Web services such as the standard WS-BPEL is their lack of formal semantics and compositionality. The BP-calculus [3] is a π -calculus based formalism encoded in WS-BPEL that has been developed with the intention to provide verification and analysis by mean of model-checking and automatic WS-BPEL code generation.

We follow this branch of research to study dynamic reconfigurations of BP-processes representing composable services. Services are self-adaptive and have to react to changes in the environment they are running in. An adaptive behaviour of a software or the need for high availability are the main motivations for dynamic reconfiguration. Dynamic reconfiguration provide a mechanism in order to meet these business requirements. A key issues for dynamic configuration requirements is the preservation of application integrity and correctness, e.g. components involved in reconfiguration must remain mutually consistent and formal verification must assert that no behavioral invariants get violated through the structural change. In [15], it has been highlighted that more research is required on dynamic reconfiguration of dependable services, and especially on its formal foundations, modelling and verification to cope with the application integrity and correctness requirement. It is the main motivation of this paper.

Contributions: Although WS-BPEL has not been designed to cope with dynamic reconfiguration, some of its features such as scopes and termination and event handlers can be used for this purpose [14]. In this paper, after having provided a complete formal specification in the BP-calculus (see Section 2) of all WS-BPEL handlers, we use this approach to model dynamic reconfiguration of complex applications involving dynamic reconfiguration in order to enable their formal analysis. Requirements that must be insured in presence of dynamic reconfiguration can then be expressed by mean of the BP-logic (see Section 2.2), opening the way to a formal verification by using existing model-checker such as the HAL Toolkit [8]. Once the verification achieved, as BP-processes are encoded in WS-BPEL, it is easy to proceed to the automatic generation of the WS-BPEL code implementing the dynamic reconfiguration. We illustrate our approach on a significant case study drawn from [14].

Related works: The work in [15] motivates the need for a formalism for the modelling and analysis of dynamic reconfiguration of dependable real-time systems. Capabilities of two home-made calculi $Web\pi_\infty$ [17] and CCS_{dp} and those of well-established formalisms namely the asynchronous π -calculus ([10], [4]) and VDM [6] are evaluated.

In [14] and [2] a case study is described using the BPMN notation [5] and formalizations by mean of the $Web\pi_\infty$ and the asynchronous π -calculus are discussed. Finally a BPMN design of the case study is translated to produce a WS-BPEL implementation. However the BPMN notation lacks of formal analysis while the $Web\pi_\infty$ formalism lacks of tools to process automatic verification. Other works exploring how dynamic configuration may be implemented in BPEL are [16] or [13]. The present work can be considered as the continuation of the π -calculus based study, taking advantage of the existence of tools and automatic generation of the corresponding WS-BPEL code.

Samples of other approaches are presented in [11] and [9]. In [11], the authors use Reo, a channel-based coordination language, to model reconfiguration as a primitive and analyze it using formal verification techniques. A full implementation of the approach in a framework that includes tools for the definition, analysis, and execution of reconfigurations, and is integrated with two execution engines for Reo, is also provided.

In [9] authors present an architecture-oriented model for dynamic reconfiguration that paves the way for the definition of ADLs that are able to address the specification of dynamic architectural characteristics of service-oriented applications. A mathematical model that can be used as a semantic domain for service-oriented architectural description languages is presented.

Organization of the paper: The rest of the article is organized as follows. In Section 2, we recall the syntax and semantics of the BP-calculus focusing on handlers and the BP-logic. We use this formalism to model dynamic reconfiguration in Section 3. In Section 4 we present the case study and formalize its functional requirements with the BP-logic to illustrate our approach. Finally, Section 5 contains conclusions and directions for future work.

Terms:		
	$t ::= x$	(variables)
	a	(names)
	u	(value)
	(t_1, \dots, t_k)	(tuple)
Correlation Set:	$\mathcal{C} ::= null \mid \mathcal{C}[\tilde{x} \leftarrow \tilde{u}]$	(correlation set)
Service:		
	$Serv ::= P$	(service)
	$[\mathcal{C} : R]c(\tilde{x}).P$	(instance spawn)
	$R ::= null \mid R P$	(running instances)
Process:		
	$P, Q ::= IG$	(input guard)
	$\bar{c}^t(M).P$	(annotated output)
	$\tau.P$	(silent action)
	$P Q$	(parallel composition)
	$P \triangleright_{c(M)} Q$	(sequential composition)
	$if\ M = N\ then\ P\ else\ P$	(conditional)
	S	(scope)
Guarded choice :		
	$IG ::= 0$	(empty process)
	$c(u).P$	(input)
	$IG + IG'$	(guarded choice)
	$[\tilde{x} \leftarrow f(M_1, \dots, M_n)]IG$	(function evaluation)
Scopes :		
	$S ::= \{\tilde{x}, P, H\}$	(scope)
	$H ::= \prod_i W_i(P_{i_1}, \dots, P_{i_n})$	(handlers)

Table1. BP-calculus Syntax

2 The BP-calculus and the BP-logic

The main motivation behind the BP-calculus is to provide a rigorous framework for the formal verification and automatic generation of WS-BPEL processes. For this reason it is very close to WS-BPEL language, providing ways to model most of BPEL constructs. Its syntax is inspired by the applied π -calculus [1]. It also allows to model correlation mechanisms that are in the core of WS-BPEL by mean of a dedicated spawn operator.

We let $\tilde{x} = (x_1, \dots, x_n)$, (resp. $\tilde{a} = (a_1, \dots, a_m)$, $\tilde{u} = (u_1, \dots, u_m)$) range over the infinite set of n-tuples of variables (resp. name, value). We denote $\tilde{x} \leftarrow \tilde{u}$ the assignment of values \tilde{u} to variables \tilde{x} . The syntax is summarized in Table 1.

We briefly explain the meaning of the most significant operators together with their intended interpretation.

$Serv$ denotes the whole defined process (service) and P, Q, \dots , processes (activities) that may occur within the main service. We syntactically distinguish between them since a whole service may be spawned due to correlation mechanisms while an activity within a process may not. This distinction is conform to the WS-BPEL specification. However, in the sequel, we often use the word “pro-

cess” for both entities. IG is an input guarded process and $IG + IG'$ behaves like a guarded choice and is intended to be translated by the `<pick>` element of WS-BPEL language.

$P \triangleright_{c(M)} Q$ expresses a sequential composition from process P passing M to Q (Q can perform actions when P has terminated). M carries binding information between processes P and Q . This construct allows to easily mimic the WS-BPEL’s element `<sequence>`.

Concerning the correlation mechanism, one of the most important element is the definition of correlation set \mathcal{C} . It is 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, $[\mathcal{C} : R]c(\tilde{x}).P$ (Instance spawn) represents an orchestration service running a process defined as $c(\tilde{x}).P$. A reception of a message M over the dedicated channel c causes a new service instance (defined as P) to be spawned. The process R represents the parallel composition of service instances already spawned, \mathcal{C} the correlation set characterizing instances. Note the way the only admitted recursion is obtained by mean of spawned services. This is to conform to the BPEL specification.

The other main feature is the definition of scopes. A scope is a wrapper for variables, a primary activity and handlers represented as contexts.

If $S ::= \{\tilde{x}, P, H\}$ is a scope, with handlers $H ::= \prod_{i=0}^n W_i(P_{i_1}, \dots, P_{i_{n_i}})$ then,

- \tilde{x} are the local variables of the scope, and P its primary activity,
- H is the parallel composition of handlers W_i . Each handler is a wrapper for a tuple of processes $\hat{P} = (P_1, \dots, P_n)$ corresponding to the activities the handler has to run when invoked.
- $W_i(P_{i_1}, \dots, P_{i_{n_i}})$ is the process obtained from the multi-hole context $W_i[\cdot]_1 \dots [\cdot]_{n_i}$ by replacing each occurrence of $[\cdot]_j$ with P_{ij} .

It has also to be noted how expressing handlers in term of primitives, simplifies a lot the specification. They are indeed considered as context where the designer has only to provide the processes to “fill” the holes and other parameters such as the kind of faults or events that are involved. This approach not only eases the specification of complex processes, but allows for an automatic mapping to well-formed BPEL-processes with a complete set of handlers.

Interested readers will find a description of operational semantics of the language in Appendix A.

2.1 Handlers

The WS-BPEL specification defines four sorts of handlers summarized as follows:

Fault Handler Faults signaled by the `<Throw>` element are caught by the fault handler. The `<catch>` element allows handling a fault specified by a fault name while the `<catchAll>` element catches any signaled fault.

Event Handler Event handlers defines the activities relative events such as an incoming message or a timeout. Event Handlers are invoked concurrently with the scope, if the corresponding event occurs.

Compensation Handler If defined the compensation handler contains the activity to be performed in the case where under certain error conditions; some previous actions need to be undone.

Termination Handler After terminating the scope's primary activity and all running event handler instances, the scope's (custom or default) termination handler is executed.

Below, the semantics associated with each of these handlers and with the whole scope:

Handlers wrappers The following semantics is widely inspired and adapted from [12]. Note that for all these handlers, $throw$, en_i , dis_i are bound names to the whole system. They are communication channels between processes.

Fault Handler Given a tuple of faults (\tilde{x}) related to a tuple of processes $\hat{P} = (P_1, \dots, P_n)$ the code for the fault handler is:

$$W_{FH}(\hat{P}) ::= en_{fh}(). \left(\sum_i (x_i(\tilde{y}). \overline{throw}^{inv} \langle \rangle | P_i). (\overline{y_1}^{inv} \langle \rangle | \overline{y_{fh}}^{inv} \langle \rangle) + dis_{fh}() \right)$$

A fault handler is enabled using en_{fh} channel. The fault handler uses a guarded sum to execute an activity P_i , associated with the triggered fault (i). After executing the associated activity, it then signals its termination to the activating process on the channel y_1 and to the scope on channel y_{fh} . If necessary, the fault handler is disabled using dis_{fh} channel. In WS-BPEL, internal faults are signaled using the `<throw>` activity.

Event Handler Given a tuple of events (\tilde{x}) related to a tuple of processes $\hat{P} = (P_1, \dots, P_n)$ the code for the event handler is:

$$W_{EH}(\hat{P}) ::= (\nu \tilde{x}) en_{eh}(). \left(\prod_i (!x_i(\tilde{y}). \overline{z_i} \langle \tilde{y} \rangle) + dis_{eh}() \mid \prod_i (z_i(\tilde{u}). P_i) \right)$$

An event handler enables itself using en_{eh} channel, then waits for a set of events on channels (\tilde{x}) each of them associated with an event. When the event occurs, the associated activity P_i is triggered by a synchronization on channel z_i . It is a typical usage of the `pick` construct. The event handler is disabled using dis_{eh} channel.

Compensation Handler Let P_1 be the scope activity and P_C the compensation activity, the compensation handler is modeled by:

$$W_{CH}(P_1, P_C) ::= en_{ch}(). \left(z(\tilde{y}). (CC(P_1, \tilde{y}) \mid \overline{throw}^{inv} \langle \rangle) + inst_{ch}(). (z(\tilde{y}). P_C \mid \overline{y_{ch}}^{inv} \langle \rangle) \right)$$

where :

$$CC(P_1, \tilde{y}) = \prod_{z' \in S_n(P_1)} \overline{z'}^{inv} \langle \tilde{y} \rangle$$

compensate children scopes (through channels in $S_n(P_1)$) of activity P_1 .

A compensation handler associated with a scope z is first installed at the beginning of the scope through an input on channel $inst_{ch}$ (\overline{y}_{ch}^{inv} signals this installation); it then process its compensation activity P_C . If the compensation handler is invoked but not installed, it signals the termination of the scope activity through channel $throw$ and performs children compensation (CC). In WS-BPEL, the compensation handler is invoked using the `<compensate>` activity.

Termination Handler To force the termination of a scope, we first disable it's event handlers and terminate primary activity and all running event handler instances. Then the custom or default `<terminationHandler>` for the scope, is run [18]. The formal model is as follows:

$$W_{TH}(P_T) ::= term(\tilde{u}).(\overline{dis_{eh}}^{inv} \langle \rangle \mid \overline{o}^{inv} \langle \tilde{y} \rangle \mid (P_T \mid \overline{throw}^{inv} \langle \rangle))$$

A termination handler is invoked by the terminating scope using channel $term$. It disables the event handler using channel dis_{eh} and terminates scope's primary activity using channel o . Then custom or default termination process P_T is run.

Scope Finally, putting all this together leads to the following semantics where the scope is represented by a hole context. Only the scope activity A must be provided by the designer.

Scope ::= $(\nu throw, en_{eh}, en_{fh}, en_{ch}, dis_{fh}, inst_{ch}, dis_{eh}, term)$

$$\begin{aligned} & (W_{EH}(\widehat{A_{eh}}) \mid W_{FH}(\widehat{A_{fh}}) \mid W_{CH}(P_1, P_C) \mid W_{TH}(P_T) \\ & \mid (\overline{en_{eh}} \langle \rangle . \overline{en_{fh}} \langle \rangle . \overline{en_{ch}} \langle \rangle)) \\ & \mid (A \mid \overline{t} \langle \rangle) \\ & \mid c().(\overline{dis_{eh}} \langle \rangle . \overline{dis_{fh}} \langle \rangle \mid \overline{inst_{ch}} \langle \rangle . \overline{term} \langle \rangle) \mid y_{eh}().y_{fh}().y_{ch}() \\ & \mid (x_z().(\overline{throw} \langle \rangle \mid \overline{dis_{fh}} \langle \rangle) + t().\overline{c} \langle \rangle)) \end{aligned}$$

- $(\overline{en_{eh}} \langle \rangle . \overline{en_{fh}} \langle \rangle . \overline{en_{ch}} \langle \rangle)$: enables handlers
- $(A \mid \overline{t} \langle \rangle)$ indicates normal termination by an output on channel t
- $c().(\overline{dis_{eh}} \langle \rangle . \overline{dis_{fh}} \langle \rangle \mid \overline{inst_{ch}} \langle \rangle . \overline{term} \langle \rangle)$: in case of normal termination, disables event and fault handlers, installs compensation handler and runs termination handler.
- $(x_z().(\overline{throw} \langle \rangle \mid \overline{dis_{fh}} \langle \rangle) + t().\overline{c} \langle \rangle)$: scope can receive a termination signal on x_z from its parents, or terminate normally by receiving a signal on t .
- $y_{eh}().y_{fh}().y_{ch}()$: channels used to indicate termination of handlers.
- P_T is the termination process.
- P_C is the compensation activity.

2.2 The BP-logic

We close these preliminaries by briefly recalling the *BP-logic*. Its syntax is a slight variant of the π -logic [7]. Its semantics is interpreted on labelled \checkmark -transition systems and it is given by the following grammar :

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

where μ is a *BP*-calculus action and χ could be μ , $\sim \mu$, or $\bigvee_{i \in I} \mu_i$ and I is a finite set. $EX\{\mu\}\phi$ is the strong next, while $EF\phi$ means that ϕ must be true sometimes in a possible future. The meaning of $EF\{\chi\}\phi$ is that validity of ϕ must be preceded by the occurrence of a sequence of actions χ . Some useful dual operators are defined as usual: $\phi \vee \phi$, $AX\{\mu\}\phi$, $\langle \mu \rangle \phi$ (weak next), $[\mu]\phi$ (dual of weak next), $AG\phi$ ($AG\{\chi\}$) (always).

Explicit interpretation of the termination predicate \checkmark is that P satisfies \checkmark iff P terminates; which is denoted $P\checkmark$. The formal definition of \checkmark is :

$$P\checkmark \text{ and } P \xrightarrow{\alpha} Q \Rightarrow Q\checkmark$$

Additional rules induced by this predicate in the operational semantics are given in Table 3 of Appendix A.

3 Formalizing dynamic reconfiguration

3.1 Dynamic reconfiguration

Dynamic reconfiguration can easily be coped with using handlers [14]. The regions to be reconfigured have to be represented by scopes so that they could contain adequate termination and event handlers, and be triggered or finished in a clean way.

So, each scope (i.e. region) will be associated with appropriate termination and event handlers. These handlers allow the the transition phase to take place: The new region has to be triggered by the event handler while the old region will be then terminated by the termination handler. Event handlers run in parallel with the scope body; so the old region can be terminated separately while the event handler brings the new region into play.

This is formalized by modeling the system *Workflow* as a sequence of an *entering* region (R_{enter}), followed by the *transition region* (TR), itself followed by the *finishing* region (R_{finish}). Thus,

$$Workflow = R_{enter} \triangleright TR \triangleright R_{finish}$$

TR is modeled as a scope with an event handler $W_{EH}(P_{E_1})$. In case the *change configuration* event is triggered, the scope corresponding to the new region is launched, while the first activity corresponding to the old region is terminated by calling its termination handler. Let R_{old} and R_{new} be respectively the old and the new region, both in TR . Thus TR is modeled as follows:

$$\begin{aligned}
TR &= \{\tilde{x}, R_{old}, H_1\} \text{ where} \\
H_1 &= W_{FH}(P_{F_1})|W_{EH}(P_{E_1})|W_{CH}(P_{C_1})|W_{TH}(P_{T_1}) \\
P_{E_1} &= \text{change}().R_{new} \\
R_{new} &= \{\tilde{y}, P_{new}, H_2\}. \\
H_2 &= W_{FH}(P_{F_2})|W_{EH}(P_{E_2})|W_{CH}(P_{C_2})|W_{TH}(P_{T_2}) \\
P_{T_2} &= \text{term}().R_{old}
\end{aligned}$$

P_{F_i} , P_{E_i} , P_{C_i} and P_{T_i} are activities respectively associated with fault, event, compensation and termination handlers of the i^{th} scope. The main activity of the transition region TR is the old region R_{old} and is processed first, unless the *change* event is triggered, in which case the new region is processed. P_{E_1} is the activity associated with the event handler in charge of triggering R_{new} by mean of channel *change*. The charge of terminating the old region is devoted to process $W_{TH}(P_{T_1})$. R_{new} is itself a scope with a main activity P_{new} and handlers H_2 .

This definition acts as a template for any dynamic reconfiguration scheme. Designers need only to fulfil holes, represented by processes P_X in each handler, to adapt it to their needs.

3.2 Expressing Requirements

Requirements can be expressed as structural and/or behavioral properties that ensure the system's consistency. They are invariants of the system and one must ensure that they are not violated. One must ensure, for instance, that whatever is the used procedure, the result is the same and the logical processing order is respected. This is an example of structural and behavioral invariant since it makes assumptions on the state of the system. We may express this property by the following generic formula where x and y are channels that respectively trigger entering and finishing regions:

$$AG\{\{x().R_{enter}\}true \wedge EF\{y().R_{finish}\}true\}$$

The $EF\{\chi\}\phi$ operator of the BP-logic allows us to express precedence properties. If R_1 and R_2 are some regions that must be processed in this order, one can formalize it like this:

$$EF\{\{x()R_1\}true\} \{y()R_2\}true$$

One may be interested in termination of processes. In this case the \checkmark operator is useful: $R\checkmark$.

4 The Case Study

In order to illustrate our approach, we use here the same case study presented in [14]. This case study describes dynamic reconfiguration of an office workflow for order processing that is commonly found in large and medium-sized organizations.

4.1 Dynamic reconfiguration of office workflow

A given organisation handles its orders from existing customers using a number of activities arranged according to the following procedure:

1. **Order Receipt:** an order for a product is received from a customer.
2. **Evaluation:** An inventory check on the availability of the product and a credit check on the customer are performed that use external services. If both the checks are positive, the order is accepted for processing; otherwise the order is rejected.
3. **Rejection:** if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.
4. If the order is to be processed, the following two activities are performed concurrently:
 - (a) **Billing:** the customer is billed for the total cost of the goods ordered plus shipping costs.
 - (b) **Shipping:** the goods are shipped to the customer.
5. **Archiving:** the order is archived for future reference.
6. **Confirmation:** a notification of successful completion of the order is sent to the customer.

In addition, for any given order, **Order Receipt** must precede **Evaluation**, which must precede **Rejection** or **Billing** and **Shipping**.

After some time, managers decide to change the order processing procedure, so that **Billing** is performed before **Shipping** (instead of performing the two activities concurrently). During the transition interval from one procedure to the other, some requirements (invariants) must be met and are presented in Section 4.3.

4.2 The model in BP-calculus

The R_{enter} region contains order reception and evaluation(*Creditcheck* and *InventoryCheck* operations) and is not detailed here. In the same manner, the R_{finish} region contains *Archiving* and *Confirmation* and are not detailed here. We focus on the transition region TR .

To model TR , we only need to provide the structure of processes P_{new} , R_{old} , P_{T_1} and P_{T_2} to complete the template designed in Section 3.1.

$$\begin{aligned} R_{old} &= \text{BillShip}().(\overline{\text{Bill}} \text{ customer, order} \mid \overline{\text{Ship}} \text{ customer, order}) \\ P_{T_1} &= \text{term}_1() \\ P_{new} &= \text{BillShip}().\overline{\text{term}_1}.(\overline{\text{Bill}} \text{ customer, order} \triangleright \overline{\text{Ship}} \text{ customer, order}) \\ P_{T_2} &= \text{term}_2() \end{aligned}$$

Regions are invoked using the *BillShip* channel. In the old region, billing and shipping are processed concurrently, while in the new region this is done sequentially. When the new region is invoked, it begins disabling the old region by invoking its termination Handler using channel term_1 .

4.3 Formalizing requirements

Key concerns raised from dynamic configuration and discussed in introduction (Section 1) can be formalized in BP-logic. Concerning the case study some of its requirements have been pointed out in [14]. They could be stated as follows:

1. The result of the billing and shipping process for any given order should not be affected by the change in procedure.
2. All orders accepted after the change in procedure must be processed according to the new procedure.

The first requirement means that whatever region chosen, an order is billed, shipped then archived. We can model this as follows:

$$AG\{\{Billship()\}true \wedge EF\{\overline{Bill\ order}\}true \wedge EF\{\overline{Ship\ order}\}true\}$$

The last requirement means that after a signal has been received on channel *change*, the termination handler of the old region is invoked. This may be modeled by the following formula:

$$AG\{\{change()\}true \wedge EF\{\overline{term_1}\}true\}$$

In order to check the termination of the whole process the formula is as follows:

$$Workflow\checkmark$$

5 Discussion and future work

In this work, we have shown a high-level approach for modeling and verifying dynamic reconfigurations of WS-BPEL services. We have expressed their requirements in the BP-logic in order to verify them with reliable existing tools such as Mobility Workbench or the HAL Toolkit.

For the future works, we are currently working on the implementation of a complete tool for analysis and verification of WS-BPEL specifications by means of the BP-calculus and the BP-logic. Another long term objective is to formally prove that our language allows for a sound automatic generation of WS-BPEL code.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th Symposium on Principles of Programming Languages*, page 104115. ACM Press, 2001.
2. F. Abouzaid, A. Bhattacharyya, N. Dragoni, JS. Fitzgerald, M. Mazzara, and M. Zhou. A case study of workflow reconfiguration: Design, modelling, analysis and implementation. School of computing science technical report series, Newcastle upon Tyne: School of Computing Science, University of Newcastle upon Tyne, 2011.

3. 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.
4. G. Boudol. Asynchrony and the π calculus. Technical Report 1702, INRIA Sophia-Antipolis, 1992.
5. BPMN. Bpmn - business process modeling notation. '<http://www.bpmn.org/>.
6. D. and C.B. Jones. *Lecture Notes in Computer Science*, volume 61, chapter The Vienna Development Method: The Meta-Language. Springer, 1978.
7. G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. Verifying mobile processes in the hal environment,. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV'98*, pages 511–515, 1998.
8. 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.
9. J.L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 70–85, Berlin, Heidelberg, 2010. Springer-Verlag.
10. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *European Conference on ObjectOriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.
11. C. Krause, Z. Maraikar, A. Lazovik, and F.Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23 – 36, 2011.
12. R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2007.
13. M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya. Design, modelling and analysis of a workflow reconfiguration. In *PNSE 2011, Petri Nets and Software Engineering (INVITED TALK)*, Newcastle, UK, 2011.
14. M. Mazzara, F. Abouzaid, N Dragoni, and A. Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations - a process algebra perspective. In *Proceedings of the 8th International Workshop on Web Services and Formal Methods (WS-FM'11)*,., Clermont-Ferrand, France, 2011.
15. M. Mazzara and A. Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In *In Third International Conference on Dependability (DEPEND 2010)*, Venice/Mestre, Italy, 2010. IEEE Computer Society.
16. M. Mazzara, N. Dragoni, and M. Zhou. Implementing workflow reconfiguration in ws-bpel. *Journal of Internet Services and Information Security (JISIS)*, to appear, 2012.
17. M. Mazzara and I. Lanese. Towards a unifying theory for web services composition. In *WS-FM*, pages 257–272, 2006.
18. 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.

A Operational semantics of the BP-calculus

The operational semantics of the BP-calculus is a labelled transition system generated by inference rules given in Table 2.

OPEN	$\frac{P \xrightarrow{\bar{a}(u)} P' \quad a \neq u}{\{u, P, \emptyset\} \xrightarrow{\bar{a}(u)} P'}$	CLOSE	$\frac{P \xrightarrow{a(u)} P' \quad Q \xrightarrow{\bar{a}(u)} Q' \quad u \notin fn(P)}{P Q \xrightarrow{\tau} \{u, P' Q', \emptyset\}}$
RES	$\frac{P \xrightarrow{\alpha} P' \quad n \notin fn(\alpha) \cup bn(\alpha)}{\{n, P, \emptyset\} \xrightarrow{\alpha} \{n, P', \emptyset\}}$	TAU	$\frac{}{\tau.P \xrightarrow{\tau} P}$
OUT	$\frac{}{\bar{c}^t \langle M \rangle . P \xrightarrow{\bar{c}(M)} P}$	IN	$\frac{}{c(x).P \xrightarrow{c(M)} P\{M/x\}}$
PAR	$\frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P Q \xrightarrow{\alpha} P' Q}$	SYNC	$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P Q \xrightarrow{\tau} P' Q'}$
CHOICE	$\frac{IG_i \xrightarrow{\bar{c}_i^t} P_i \quad i \in \{1, 2\}}{IG_1 + IG_2 \xrightarrow{\alpha} P_i}$	DEF	$\frac{P\{\bar{y}/\bar{x}\} \xrightarrow{\alpha} P' \quad A(\bar{x}) = P}{A(\bar{x}) \xrightarrow{\alpha} P'}$
SCO	$\frac{P \xrightarrow{\alpha} P'}{\{x, P, H\} \xrightarrow{\alpha} \{x, P', H\}}$	HAN	$\frac{H \xrightarrow{\alpha} H'}{\{x, P, H\} \xrightarrow{\alpha} \{x, P, H'\}}$
SPAR	$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{\{x_1, P, H_1\} \{x_2, Q, H_2\} \xrightarrow{\tau} \{x_1, P', H_1\} \{x_2, Q', H_2\}}$		
IFT-M	$\frac{P \xrightarrow{\alpha} P' \quad M=N}{if(M=N) \text{ then } P \text{ else } Q \xrightarrow{\alpha} P'}$	IFF-M	$\frac{Q \xrightarrow{\alpha} Q' \quad M \neq N}{if(M=N) \text{ then } P \text{ else } Q \xrightarrow{\alpha} Q'}$
EVAL	$\frac{\bar{M} = f(M_1, \dots, M_n) \quad P\{\bar{M}/\bar{x}\} \xrightarrow{\alpha} P'}{[\bar{x} \leftarrow f(M_1, \dots, M_n)]P \xrightarrow{\alpha} P'}$		
SEQ1	$\frac{P \xrightarrow{\alpha} P'}{P \triangleright_{c(M)} Q \xrightarrow{\alpha} P' \triangleright_{c(M)} Q}$	SEQ2	$\frac{P \xrightarrow{\bar{c}(M)} 0 \quad Q \xrightarrow{c(M)} Q'}{P \triangleright_{c(M)} Q \xrightarrow{\tau} Q'}$
C-SP1	$\frac{Serv_0 \xrightarrow{\alpha} Serv'_0 \quad Serv = c_A(\bar{x}).A(\bar{y})}{[C:R]Serv_0]c_A(\bar{x}).A(\bar{y}) \xrightarrow{\alpha} [C:R]Serv'_0]c_A(\bar{x}).A(\bar{y})}$		
C-SPT	$\frac{createInstance(M) = true \quad [\bar{z} \leftarrow \bar{u}] = correlationPart(M)}{[null:0]c_A(\bar{x}).A(\bar{y}) \xrightarrow{c_A(M)} [[\bar{z} \leftarrow \bar{u}]:A(\bar{u})]c_A(\bar{x}).A(\bar{y})}$		
C-SPF	$\frac{createInstance(M) = true \quad [\bar{z} \leftarrow \bar{u}] = corrPart(M) \quad [\bar{z} \leftarrow \bar{u}] \notin C \quad Serv = c_A(\bar{x}).A(\bar{y})}{[C:R]c_A(\bar{x}).A(\bar{y}) \xrightarrow{c_A(M)} [C, [\bar{z} \leftarrow \bar{u}]:R]A(\bar{u})]c_A(\bar{x}).A(\bar{y})}$		

Table2. Operational semantics of the BP-calculus.

TER1	$\overline{0\checkmark}$	TER2	$\frac{P\checkmark}{\tau.P\checkmark}$	TER3	$\frac{P\checkmark}{c(\bar{u}).P\checkmark}$	TER4	$\frac{P\checkmark}{\bar{c}^t \langle M \rangle . P\checkmark}$
TER5	$\frac{P\checkmark \quad Q\checkmark}{(P \triangleright_{c(M)} Q)\checkmark}$	TER6	$\frac{P\checkmark \quad Q\checkmark}{(P Q)\checkmark}$	TER7	$\frac{P\checkmark}{\{n, P, \emptyset\}\checkmark}$	TER8	$\frac{P\checkmark}{\{\bar{x}, P, H\}\checkmark}$

Table3. Operational semantics induced by \checkmark predicate.