

# A Formal Semantics for the WS-BPEL Recovery Framework The $\pi$ -Calculus Way

Nicola Dragoni<sup>1</sup> and Manuel Mazzara<sup>2</sup>

<sup>1</sup> DTU Informatics, Technical University of Denmark, Denmark  
`ndra@imm.dtu.dk`

<sup>2</sup> School of Computing Science, Newcastle University, UK  
`manuel.mazzara@newcastle.ac.uk`

**Abstract.** While current studies on Web services composition are mostly focused — from the technical viewpoint — on standards and protocols, this work investigates the adoption of formal methods for dependable composition. The Web Services Business Process Execution Language (WS-BPEL) — an OASIS standard widely adopted both in academic and industrial environments — is considered as a touchstone for concrete composition languages and an analysis of its ambiguous Recovery Framework specification is offered. In order to show the use of formal methods, a precise and unambiguous description of its (simplified) mechanisms is provided by means of a conservative extension of the  $\pi$ -calculus. This has to be intended as a well known case study providing methodological arguments for the adoption of formal methods in software specification. The aspect of verification is not the main topic of the paper but some hints are given.

## 1 Introduction

Service Oriented Architectures and the related paradigm are modern attempts to cope with old problems connected to Business-to-Business (B2B) and information interchange. Many implementations of this paradigm are possible and the so called Web services look to be the most prominent, mainly because the underlying architecture is already there; it is simply the web which has been extensively used in the last 15 years and where we can easily exploit HTTP [21], XML [5], SOAP [8] and WSDL [3]. The World Wide Web provides a basic platform for the interconnection on a point-to-point basis of different companies and customers but one of the B2B complications is the management of causal interactions between different services and the way in which the messages between them need to be handled (e.g., not always in a sequential way). This area of investigation is called composition, *i.e.*, the way to build complex services out of simpler ones [4]. These days, the need for workflow technology is becoming quite evident and the positive aspect is that we had investigated this technology for decades and we also have excellent modeling tools providing verification features that are grounded in the very active field of concurrency theory research.

### 1.1 BPEL and Its Ambiguous Specification

Several organizations worked on composition proposals. The most important in the past have been IBM’s WSFL [1] and Microsoft’s XLANG [2]. These two have then converged into Web Services Business Process Execution Language [18] (BPEL for short) which is presently an OASIS standard and, given its wide adoption, it will be used as a touchstone for composition languages in this paper. BPEL allows workflow-based composition of services. In the committee members’ words the aim is “*enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks*”. The problem with BPEL was that the earlier versions of the language were not very clear, the specification was huge and many points confusing, especially in relation to the Recovery Framework (RF) and the interactions between different mechanisms (fault handlers and compensation handlers). BPEL indeed represents a business tradeoff where not necessarily all the single technical choices have been made considering all the available options. Although in the final version of the specification (which is lighter and cleaner) fault handling during compensation has been simplified, we strongly believe that the sophisticated mechanism of recovery still needs a clarification.

### 1.2 Contribution of the Paper

In this paper we aim to reduce this ambiguity providing an *easily readable formal semantics* of the BPEL Recovery Framework (BPEL RF for short). This goal requires at least two different contributions:

1. a *formal semantics* of the framework, focusing on its essential mechanisms
2. an *easily readable* specification of these mechanisms

We provide both contributions following a “ $\pi$ -calculus way”, that is using the  $\pi$ -calculus as formal specification language. It is worth noting that here the actual challenge is to provide not only a formal semantics for the BPEL RF but also an *easily readable* specification. Indeed, other attempts might be found in literature providing the first contribution only. For instance, in [12] such encoding has been proposed by one of the authors. However, one of the unsatisfactory aspects about that encoding is that it is hardly readable and complex. The actual challenge here is to reduce such complexity while keeping a formal and rigorous approach. As a result, in this paper we contribute with a better understanding of how the BPEL RF works. Moreover, the case study allows us to show the real power of the  $\mathbf{web}\pi_\infty$  calculus (*i.e.*, the  $\pi$ -calculus based formal language exploited for the mentioned encoding) not only in terms of simplicity of the resulting BPEL specification, but also sketching how  $\mathbf{web}\pi_\infty$  can contribute to the implementation of real orchestration engines.

Finally, we would like to stress that different formal models might be chosen for this goal. As discussed in the next section, our choice is primarily motivated by the “foundational feature” of the  $\pi$ -calculus, namely *mobility*, *i.e.* the possibility of transmitting channel names that will be, in turn, used by any receiving

process. It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, in the general case, we have the strong opinion that mobility is an essential feature that composition languages should exhibit [13]. This aspect will be better discussed in section 2.

**Outline.** The paper is organized as follows. Section 2 will discuss the rationale behind our “ $\pi$ -calculus way” choice, briefly motivating why the  $\pi$ -calculus could be considered a formal foundation for dependable Web services composition. Section 3 will present  $\mathbf{web}\pi_\infty$  discussing its syntax and semantics. Sections 4 and 5 will contribute with a clarification of the BPEL RF semantics. In particular, Section 4 will show how it appears in the original (ambiguous) specification, and Section 5 will propose the actual simplification and formal specification. Section 6 will add some conclusive remarks.

## 2 The $\pi$ -Calculus Way to Dependable Composition

The need for formal foundation has been discussed widely in the last years, although many attempts to use formal methods in this setting have been speculative. Some communities, for example, criticized the process algebra options [19] promoting the Petri nets choice. The question here is whether we need a formal foundation and, if that is the case, which kind of formalism we need. While sequential computation has well established foundations in the  $\lambda$ -calculus and Turing machines, when it comes to concurrency things are far from being settled. The  $\pi$ -calculus ([17] and [16]) emerged during the eighties as a theory of mobile systems providing a conceptual framework for expressing them and reasoning about their behavior. It introduces mobility generalizing the channel-based communication of CCS by allowing channels to be passed as data through rendezvous over other channels. In other words, it is a model for prescribing (specification) and describing (analysis) concurrent systems consisting of agents which mutually interact and in which the communication structure can dynamically evolve during the execution of processes. Here, a communication topology is intended as the linkage between processes which indicates who can communicate with whom. Thus, changing the communication links means, for a process, moving inside this abstract space of linked entities.

A symmetry between  $\lambda$ -calculus and  $\pi$ -calculus could be suggested and the option to build concurrent languages (and so workflow languages as well) on a formal basis could actually make sense. It has indeed been investigated in many works, even in the BPEL context. But, while formal methods are expected to bring mathematical precision to the development of computer systems (providing precise notations for specification and verification), so far BPEL — despite having been subject of a number of formalizations (for example [10], [7] and [22]) — has not yet been proved to be built on an exact and specific mathematical

model, including process algebras (this argument has been carefully developed in [13]). Thus, we do not have any conceptual and software tools for analysis, reasoning and software verification. If we are not able to provide this kind of tools, any hype about mathematical rigor becomes pointless.

It is also worth noting that, although many papers use the term  $\pi$ -calculus and process algebra interchangeably, there is a difference between them. Algebra is a mathematical structure with a set of values and a set of operations on the values. These operations enjoy algebraic properties such as commutativity, associativity, idempotency, and distributivity. In a typical process algebra, processes are values and parallel composition is defined to be a commutative and associative operation on processes. The  $\pi$ -calculus is an algebra but it differs from previous models for concurrency precisely for the fact that it includes a notion of mobility, i.e. the possibility of transmitting channel names that will be, in turn, used by receiving processes. This allows a sort of dynamic reconfiguration with the possibility of creating (and deleting) processes through the alteration of the process topology (although it can be argued that, even if the link to a process disappears, the process itself disappears only from “an external point of view”).

The  $\pi$ -calculus looks interesting because of its treatment of component bindings as first class objects, which enables this dynamic reconfiguration to be expressed simply. So, the question now is: do we need this additional feature of the  $\pi$ -calculus or should we restrict our choice to models, like CCS, without this notion of mobility? Why all this hype over the  $\pi$ -calculus and such a rare focus on its crucial characteristic? We have the strong opinion that mobility is an essential feature that composition languages should exhibit. Indeed, while in some scenarios services can be selected already at design-time, in others some services might only be selected at runtime and this selection has then to be propagated to different parties. This phenomenon is called link passing mobility and it is properly approached in [6].

It is worth noting that in the specific contribution of this paper this feature is not really exploited or totally necessary since the modeled mechanisms requested we pay more attention to process synchronization and concurrency than to full mobility. This aspect has been instead essential in the full formalization of BPEL. In [13] it has been shown how it plays an important role in the encoding of interactions of the kind request-response. Indeed, in that case the invoker must send a channel name to be used then to return the response. This is a typical case of the so called output capability of the  $\pi$ -calculus, i.e. a received name is used as the subject of outputs only. The full input capability of the  $\pi$ -calculus — i.e. when a received name is used also as the subjects of inputs — has been not exploited in the BPEL encoding (and neither it is in this work). Indeed in [13] a specific well-formedness constraint imposes that “*received names cannot be used as subjects of inputs or of replicated inputs*”. Thus, at the present moment we remain agnostic regarding the need of the  $\pi$ -calculus input capability in the description of BPEL mechanism. We realize that this admission could be an argument for discussing again the choice of the original model.

## 2.1 Our Approach

WS-standards for dependability only concerns SOAP when employed as an XML messaging protocol (e.g. OASIS WS-Reliability and WS-Security), *i.e.*, at the message level. However, things are more complicated than this since loosely coupled components like Web services, being autonomous in their decisions, may refuse requests or suspend their functionality without notice, thus making their behavior unreliable to other activities. Henceforth, most of the web languages also include the notion of loosely coupled transaction – called *web transaction* [11] in the following – as a unit of work involving loosely coupled activities that may last long periods of time. These transactions, being orthogonal to administrative domains, have the typical atomicity and isolation properties relaxed, and instead of assuming a perfect roll-back in case of failure, support the explicit programming of compensation activities. Web transactions usually contain the description of three processes: *body*, *failure handler*, and *compensation*. The failure handler is responsible for reacting to events that occur during the execution of the body; when these events occur, the body is blocked and the failure handler is activated. The compensation, on the contrary, is installed when the body commits; it remains available for outer transactions to require some undo of previously performed actions. BPEL also uses this approach.

Our approach to recovery is instead described in [13], where it has been shown that different mechanisms for error handling are not necessary and the BPEL semantics has been presented in terms of  $\mathbf{web}\pi_\infty$ , which is based on the idea of event notification as the unique error handling mechanism. This result allows us to extend any semantic considerations about  $\mathbf{web}\pi_\infty$  to BPEL.  $\mathbf{web}\pi_\infty$  (originally in [14]) has been introduced to investigate how process algebras can be used as a foundation in this context. It is a simple and conservative extension of the  $\pi$ -calculus where the original algebra is augmented with an operator for asynchronous events raising and catching in order to enable the programming of widely accepted error handling techniques (such as long running transactions and compensations) with reasonable simplicity. We addressed the problem of composing services starting directly from the  $\pi$ -calculus and considering this proposal as a foundational model for composition simply to verify statements regarding any mathematical foundations of composition languages and not to say that the  $\pi$ -calculus is more suitable than other models (such as Petri nets) for these purposes. The calculus is presented in detail in section 3 while in section 4 and 5 it is showed how it can be useful to clarify the BPEL RF semantics.

## 3 The Composition Calculus

In this section we present a proposal to cope with the issues presented in section 2. Although  $\mathbf{web}\pi_\infty$  is ambitious, for sure we do not pretend to solve all the problems and to give the ultimate answer to all the questions. Giving all the details about the language and its theory is beyond the scope of this paper which is giving a brief account about how  $\mathbf{web}\pi_\infty$  can be considered in the overall scenario of formal methods for dependable Web services. You can find all the relevant details in some previous work, especially in [12], [13] and [15].

### 3.1 Syntax

The syntax of  $\mathbf{web}\pi_\infty$  processes relies on a countable set of *names*, ranged over by  $x, y, z, u, \dots$ . Tuples of names are written  $\tilde{u}$ . We intend  $i \in I$  with  $I$  a finite non-empty set of indexes.

$$\begin{array}{ll}
P ::= & \mathbf{0} & \text{(nil)} \\
& | \bar{x}\tilde{u} & \text{(output)} \\
& | \sum_{i \in I} x_i(\tilde{u}_i).P_i & \text{(alternative composition)} \\
& | (x)P & \text{(restriction)} \\
& | P | P & \text{(parallel composition)} \\
& | !x(\tilde{u}).P & \text{(guarded replication)} \\
& | \langle P ; P \rangle_x & \text{(workunit)}
\end{array}$$

A process can be the inert process  $\mathbf{0}$ , an output  $\bar{x}\tilde{u}$  sent on a name  $x$  that carries a tuple of names  $\tilde{u}$ , an alternative composition consisting of input guarded processes that consumes a message  $\bar{x}_i\tilde{w}_i$  and behaves like  $P_i\{\tilde{w}_i/\tilde{u}_i\}$ , a restriction  $(x)P$  that behaves as  $P$  except that inputs and messages on  $x$  are prohibited, a parallel composition of processes, a replicated input  $!x(\tilde{u}).P$  that consumes a message  $\bar{x}\tilde{w}$  and behaves like  $P\{\tilde{w}/\tilde{u}\} | !x(\tilde{u}).P$ , or a workunit  $\langle P ; Q \rangle_x$  that behaves as the *body*  $P$  until an abort  $\bar{x}$  is received and then behaves as the *event handler*  $Q$ .

Names  $x$  in outputs, inputs, and replicated inputs are called *subjects* of outputs, inputs, and replicated inputs, respectively. It is worth to notice that the syntax of  $\mathbf{web}\pi_\infty$  processes simply augments the asynchronous  $\pi$ -calculus with workunit process. The input  $x(\tilde{u}).P$ , restriction  $(x)P$  and replicated input  $!x(\tilde{u}).P$  are binders of names  $\tilde{u}$ ,  $x$  and  $\tilde{u}$  respectively. The scope of these binders is the process  $P$ . We use the standard notions of  $\alpha$ -equivalence, *free* and *bound names* of processes, noted  $\text{fn}(P)$ ,  $\text{bn}(P)$  respectively.

### 3.2 Semantics

We give the semantics for the language in two steps, following the approach of Milner [16], separating the laws that govern the static relations between processes from the laws that rule their interactions. The first step is defining a static structural congruence relation over syntactic processes. A structural congruence relation for processes equates all agents we do not want to distinguish. It is introduced as a small collection of axioms that allow minor manipulation on the processes' structure. This relation is intended to express some intrinsic meanings of the operators, for example the fact that parallel is commutative. The second step is defining the way in which processes evolve dynamically by means of an operational semantics. This way we simplify the statement of the semantics just closing with respect to  $\equiv$ , *i.e.*, closing under process order manipulation induced by structural congruence.

**Definition 1.** *The structural congruence  $\equiv$  is the least congruence satisfying the Abelian Monoid laws for parallel and summation (associativity, commutativity and  $\mathbf{0}$  as identity) closed with respect to  $\alpha$ -renaming and the following axioms:*

1. *Scope laws:*

$$\begin{aligned} (u)\mathbf{0} &\equiv \mathbf{0}, & (u)(v)P &\equiv (v)(u)P, \\ P \mid (u)Q &\equiv (u)(P \mid Q), & \text{if } u \notin \text{fn}(P) \\ \langle (z)P ; Q \rangle_x &\equiv (z)\langle P ; Q \rangle_x, & \text{if } z \notin \{x\} \cup \text{fn}(Q) \end{aligned}$$

2. *Workunit laws:*

$$\begin{aligned} \langle \mathbf{0} ; Q \rangle_x &\equiv \mathbf{0} \\ \langle \langle P ; Q \rangle_y \mid R ; R' \rangle_x &\equiv \langle P ; Q \rangle_y \mid \langle R ; R' \rangle_x \end{aligned}$$

3. *Floating law:*

$$\langle \bar{z}\tilde{u} \mid P ; Q \rangle_x \equiv \bar{z}\tilde{u} \mid \langle P ; Q \rangle_x$$

The scope laws are standard while novelties regard workunit and floating laws. The law  $\langle \mathbf{0} ; Q \rangle_x \equiv \mathbf{0}$  defines committed workunit, namely workunit with  $\mathbf{0}$  as body. These ones, being committed, are equivalent to  $\mathbf{0}$  and, therefore, cannot fail anymore. The law  $\langle \langle P ; Q \rangle_y \mid R ; R' \rangle_x \equiv \langle P ; Q \rangle_y \mid \langle R ; R' \rangle_x$  moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children ones by means of names. The law  $\langle \bar{z}\tilde{u} \mid P ; Q \rangle_x \equiv \bar{z}\tilde{u} \mid \langle P ; Q \rangle_x$  floats messages outside workunit boundaries. By this law, messages are particles that independently move towards their inputs. The intended semantics is the following: if a process emits a message, this message traverses the surrounding workunit boundaries until it reaches the corresponding input. In case an outer workunit fails, recoveries for this message may be detailed inside the handler processes.

The dynamic behavior of processes is defined by the reduction relation where we use the shortcut:

$$\langle P ; Q \rangle \stackrel{\text{def}}{=} (z)\langle P ; Q \rangle_z \text{ where } z \notin \text{fn}(P) \cup \text{fn}(Q)$$

**Definition 2.** *The reduction relation  $\rightarrow$  is the least relation satisfying the following axioms and rules, and closed with respect to  $\equiv$ ,  $(x)_-$ ,  $- \mid -$ , and  $\langle - ; Q \rangle_z$ :*

$$\begin{aligned} & \text{(COM)} \\ & \bar{x}_i \tilde{v} \mid \sum_{i \in I} x_i(\tilde{u}_i).P_i \rightarrow P_i\{\tilde{v}/\tilde{u}_i\} \\ & \text{(REP)} \\ & \bar{x} \tilde{v} \mid !x(\tilde{u}).P \rightarrow P\{\tilde{v}/\tilde{u}\} \mid !x(\tilde{u}).P \\ & \text{(FAIL)} \\ & \bar{x} \mid \langle \prod_{i \in I} \sum_{s \in S} x_{is}(\tilde{u}_{is}).P_{is} \mid \prod_{j \in J} !x_j(\tilde{u}_j).P_j ; Q \rangle_x \rightarrow \langle Q ; \mathbf{0} \rangle \end{aligned}$$

$$\text{where } J \neq \emptyset \vee (I \neq \emptyset \wedge S \neq \emptyset)$$

Rules (COM) and (REP) are standard in process calculi and models input-output interaction and lazy replication. Rule (FAIL) models workunit failures: when a unit abort (a message on a unit name) is emitted, the corresponding body is terminated and the handler activated. On the contrary, aborts are not possible if the transaction is already terminated (namely every thread in the body has completed its own work), for this reason we close the workunit restricting its name.

Interested readers may find all the definitions and proofs with an extensive explanation for the extensional semantics, the notions of barb, process contexts and barbed bisimulation in [13]. Definitions for Labelled Semantics, asynchronous bisimulation, labelled bisimilarity and the proof that it is a congruence are also present. Finally, results relating barbed bisimulation and asynchronous labeled bisimulation as well as many examples are discussed. A core BPEL is encoded in  $\text{web}\pi_\infty$  and a few properties connected to this encoding are proved for it.

## 4 A Case Study: The BPEL RF

One of the unsatisfactory things about the encoding of the BPEL RF we presented in [12] is that it was hardly readable for humans. The goal was to capture in that encoding all the hidden details of the BPEL semantics and working out the full theory also for verification purpose. But surely we lost something in readability since the target for that encoding were not humans but machines. Many people who approached our work justified their problems in understanding the encoding claiming that was exactly the proof of the BPEL recovery framework complexity. This is definitely true but, in order to be really useful, that work needs to be understandable also to non-specialists (and humans in general). With the goal of better understanding how the BPEL RF works, in this section we analyze a case study where  $\text{web}\pi_\infty$  shows its power. We will firstly report the description of the mechanisms following the original BPEL specification, then we will consider a simplification of the actual mechanisms giving a simplified semantics and a simplified explanation. In this way some details will be lost but we will improve readability. The first simplification is considering only the case in which a single handler exists for each of the three different type (fault, compensation and event). Furthermore, we do not consider interdependencies between the mechanisms: default handlers with automatic compensation of inner scope. This study is an integration of what done before in [12] and [15]. The semantics provided is not the one implemented by the engines supporting BPEL, we have already given a formalization for the Oracle BPEL Manager in [13]. While in [12] you can find a complete description, here we want to focus only on the essence of the single mechanisms to understand at which stage of the execution they play their role and in which way.

### 4.1 Details from the BPEL Specification

Instead of assuming a perfect roll-back in case of failure, BPEL supports in its RF the notion of the so-called *loosely coupled transactions* and the explicit programming of compensation activities. This kind of transactions lasts long periods (atomicity needs to be relaxed wrt ACIDity), crosses administrative domains (isolation needs to be relaxed) and possibly fails because of services unavailability etc... They usually contain the description of three processes:

- body
- fault handler
- compensation handler

BPEL also adds the possibility to have a third kind of handler called the event handler. The whole set of activities is included in a construct called **scope** introduced as follows in the specification:

“A **scope** provides the context which influences the *execution behavior* of its enclosed activities. This behavioral context includes variables, partner links, message exchanges, correlation sets, *event handlers*, *fault handlers*, a *compensation handler*, and a termination handler [...]

Each **scope** has a required primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth. All other syntactic constructs of a **scope** activity are optional, and some of them have default semantics. The context provided by a **scope** is shared by all its nested activities.”

In the following, we report the way in which the concepts of the Recovery Framework and the need for it are motivated in [18].

### Compensation Handler

“Business processes are often of long duration. They can manipulate business data in back-end databases and line-of-business applications. Error handling in this environment is both difficult and business critical. The use of ACID transactions is usually limited to local updates because of trust issues and because locks and isolation cannot be maintained for the long periods during which fault conditions and technical and business errors can occur in a business process instance. As a result, the overall business transaction can fail or be cancelled after many ACID transactions have been committed. The partial work done must be undone as best as possible. Error handling in BPEL processes therefore leverages the concept of compensation, that is, *application-specific activities that attempt to reverse the effects of a previous activity that was carried out as part of a larger unit of work that is being abandoned*. There is a history of work in this area regarding the use of Sagas and open nested transactions. BPEL provides a variant of such a compensation mechanism by providing the ability for flexible control of the reversal. BPEL achieves this by providing the ability to define fault handling and compensation in an application-specific manner, in support of Long-Running Transactions (LRT's) [...] *BPEL allows scopes to delineate that part of the behavior that is meant to be reversible* in an application-defined way by specifying a compensation handler. Scopes with compensation and fault handlers can be nested without constraint to arbitrary depth.[...] A compensation handler can be invoked by using the `compensateScope` or `compensate` (together referred to as the “compensation activities”). A compensation handler for a scope **MUST** be made *available for invocation only when the scope completes successfully*. *Any attempt to compensate a scope, for which the compensation handler either has not been installed or has been installed and executed, MUST be treated as executing an empty activity.* [...]”

### Fault Handler

“Fault handling in a business process can be thought of as *a mode switch from the normal processing in a scope*. Fault handling in BPEL is designed to be treated as “reverse work” in that its aim is *to undo the partial and unsuccessful work of a scope in which a fault has occurred*. The completion of the activity of a fault handler, even when it does not rethrow the handled fault, is not considered successful completion of the attached scope. *Compensation is not enabled for a scope that has had an associated fault handler invoked*.”

Explicit fault handlers, if used, attached to a scope provide a way to define a set of custom fault-handling activities, defined by catch and catchAll constructs. Each catch construct is defined to intercept a specific kind of fault, defined by a fault QName. An optional variable can be provided to hold the data associated with the fault. If the fault name is missing, then the catch will intercept all faults with the same type of fault data. The fault variable is specified using the faultVariable attribute in a catch fault handler. The variable is deemed to be implicitly declared by virtue of being used as the value of this attribute and is local to the fault handler. It is not visible or usable outside the fault handler in which it is declared. A catchAll clause can be added to catch any fault not caught by a more specific fault handler.”

### Event Handler

“Each scope, including the process scope, can have a set of event handlers. *These event handlers can run concurrently and are invoked when the corresponding event occurs [...]* There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times.”

## 5 Formal Semantics of a (Simplified) BPEL RF

The plain text description of these mechanisms taken from the specification should give an idea of the complexity of this framework. The main difficulty we have found at the beginning of this investigation was to clarify the basic difference between failure and compensation handlers, since many words have been spent on this but the true essence of these mechanisms has never been given in a concise and simple way. In the past we also promoted a complete explanation of the mechanisms focusing on inessential minor details. Here we want to give the basic idea explaining that failure and compensation handlers differ mainly because they play their role at different stages of computation: failure handler is responsible for reacting to signals that occur during the normal execution of the body; when these occur, the body is interrupted and the failure handler is activated. On the contrary, compensation handler is installed only when the body successfully terminates. It remains available if another activity

requires some undo of the committed activity. In some sense, failures regard “living” (not terminated) processes, while compensation is only for “successfully terminated process”. The key point regarding event handlers is instead bound to the sentence reported above: they are *invoked concurrently* to the body of a scope that meanwhile continues running. This is very different from what happens for failures that interrupt the main execution and compensations which run only after the completion of the relative body.

The difficulty of the encoding we gave in [12] lies in the nontrivial interactions between the different mechanisms and it is due to the sophisticated implicit mechanism of recovery activated when designer-defined fault or compensation handlers are absent. Indeed, in this case, BPEL provides backward compensation of nested activities on a causal dependency basis relying on two rules:

- control dependency: links and sequence define causality
- peer-scope dependency: the basic control dependency causality is reflected over peer scopes

These two rules resemble some kind of structural inductive definition, as is usually done in process algebra. It is exactly our goal to skip these details here and to clarify the semantics.

### 5.1 Syntax

Let  $(A; H)_s$  be a scope named  $s$  where  $A$  is the main activity (body) and  $H$  a handler. Both  $A$  and  $H$  have to be intended as BPEL activities coming from a subset of the ones defined in [12]. Practically, that work was limited to basic activities, structured activities and error handling. The idea now is to represent a simplified BPEL scope called  $s$  having a single handler  $H$ , so we are providing a semantics for the error handling mechanisms alternative to the previous one. For the sake of simplicity, we start considering a single handler at a time. Afterward we will consider the full scope construct. In the following subsection the formal semantics derived from  $\mathbf{web}\pi_\infty$  will be presented, here we just define the syntax giving an informal explanation.

**Definition 3 (Compensation Handler).** *We define the compensation handler as follows:*

$$(A; \mathit{COMP} s \rightarrow C)_s$$

*If  $s$  is invoked after the successful termination of  $A$ , then run the allocated compensation  $C$ .*

**Definition 4 (Fault Handler).** *We define the fault handler as follows:*

$$(A; \mathit{FAULT} f \rightarrow F)_s$$

*If  $f$  is invoked in  $A$ , then abort immediately the body  $A$  and run  $F$ .*

**Definition 5 (Event Handler).** *We define the event handler as follows:*

$$(A; \mathit{EVENT} e \rightarrow E)_s$$

*If  $e$  is invoked in  $A$  then run  $E$  in parallel while the body  $A$  continues running still listening for another event  $e$ .*

## 5.2 Semantics

The formal semantics of the three mechanisms is defined here in terms of  $\mathbf{web}\pi_\infty$ . These constructs are encoded in  $\mathbf{web}\pi_\infty$  which has a formal semantics, as a consequence the semantic of the constructs themselves is given. The continuation passing style technique is used like in [12]. Briefly,  $\llbracket A \rrbracket_y$  means that the encoding of the BPEL activity  $A$  completes with a message sent over the channel  $y$ . More details can be found also in [13]. In that work the function  $\llbracket A \rrbracket_y : A_{BPEL} \rightarrow Process$  has been used to map BPEL activities into  $\mathbf{web}\pi_\infty$  processes flagging out  $y$  to signal termination.

## 5.3 Compensation Handler

**Definition 6 (Compensation Handler).** *The semantics of the single Compensation Handler scope is defined in terms of  $\mathbf{web}\pi_\infty$  as follows:*

$$(A; COMP\ s \rightarrow C)_s \stackrel{\text{def}}{=} (y)(y')(\langle \llbracket A \rrbracket_y ; s().\llbracket C \rrbracket_{y'} \rangle_y)$$

The reader will realize that there are two new names  $y$  and  $y'$  defined at the outer level. This means that all the interactions related to this name are local to this process, *i.e.*, interferences from the outside are not allowed (they are restricted names). Then you have a workunit containing the main process and the compensation handler. Both these processes are, in turn, contained by the double brackets, which means that their encodings need to be put here. As you can see the compensation is blocked until a message on  $s$  (the name of the scope) is received and  $C$  will be available only after the successful termination of  $A$  signaled on the local channel  $y$ . This expresses exactly the fact that the compensation is available only after the successful termination of the body as required in the BPEL specification. The reason for which  $C$  is activated after the termination of  $A$  stands in the  $\mathbf{web}\pi_\infty$  rule (FAIL) which activates the workunit handler  $s().\llbracket C \rrbracket_{y'}$  when the signal  $y$  (the workunit name) is received. This name is precisely sent by  $A$  when it terminates (because of the continuation passing style encoding).

## 5.4 Fault Handler

**Definition 7 (Fault Handler).** *The semantics of the single Fault Handler scope is defined in terms of  $\mathbf{web}\pi_\infty$  as follows:*

$$(A; FAULT\ f \rightarrow F)_s \stackrel{\text{def}}{=} (f)(y)(y')(\langle \llbracket A \rrbracket_y ; \llbracket F \rrbracket_{y'} \rangle_f)$$

The fault handler has a semantic very close to the  $\mathbf{web}\pi_\infty$  workunit. For this reason the encoding here is basically an isomorphism. The handler is triggered when receiving the signal  $f$  which interrupts the normal execution of the body. Since the activation of the fault handler is internal to the scope itself, the scope name is not relevant in the right hand side.

### 5.5 Event Handler

**Definition 8 (Event Handler).** *The semantics of the single Event Handler scope is defined in terms of  $\mathbf{web}\pi_\infty$  as follows:*

$$(A; \mathit{EVENT} \ e \rightarrow E)_s \stackrel{\text{def}}{=} (e)(y)(y')(\langle \llbracket A \rrbracket_y ; \mathbf{0} \rangle_y \mid !e().\llbracket E \rrbracket_{y'})$$

The event handler is interesting. The main point here is that the body execution is not interrupted when  $e$  is received. Consider indeed that  $E$  is outside the workunit and it is triggered only by  $e$ . The handler, receiving  $e$  and activating  $E$ , will run in parallel with  $A$  without interrupting it. It is worth noting also that the presence of the replication allows  $e$  to be received many times during the execution of  $A$ , each time running a new handler. The event handler will stay active without any risk of being stopped by other scopes since all the names inside the handler are local to  $E$  (bound names) due to the way in which BPEL activities are encoded by the function  $\llbracket A \rrbracket_y$ . This is a simplification to clarify the mechanism, it actually represents a deviation from the BPEL standard where the events are not restricted in this way.

### 5.6 BPEL Scope

Now that we have understood each mechanism let us put all together. We define a scope construct including all the three handlers. Again, we consider single handlers of each type with no interactions, no default handler and no automatic compensation of inner scopes.

**Definition 9 (Full Scope Construct).** *The semantics of the full scope construct is defined in terms of  $\mathbf{web}\pi_\infty$  as follows:*

$$\begin{aligned} (A; \mathit{FAULT} \ f \rightarrow F; \mathit{EVENT} \ e \rightarrow E; \mathit{COMP} \ s \rightarrow C)_s &\stackrel{\text{def}}{=} \\ (e)(f)(y)(y')(y'')(y''')(\langle \llbracket A \rrbracket_y ; \llbracket F \rrbracket_{y'} \rangle_f & \\ \mid !e().\llbracket E \rrbracket_{y''} \mid \langle (x)x() ; s().\llbracket C \rrbracket_{y'''} \rangle_y) & \end{aligned}$$

It is worth noting that here the name  $s$  is a free global name (undefined) available to all the scopes which possibly run in parallel. The technical problem is that, in this way, the encoding is not compositional. Actually, this problem is easily fixed when the encoding is extended to the complete set of BPEL constructs, including the top level process where all the scopes are defined since there you can restrict all the names of the inner scopes. This has been done previously in [12]. The purpose of this work is just to explain in a clearer way the differences between the mechanisms of the recovery framework without presenting again the whole encoding. A synergy between this result and what we have done in [12] is left as future work.

### 5.7 Example

Let us now show an example of how this mechanism works in practice. To do this we will run a process description on the “reduction semantics machine”

of  $\mathbf{web}\pi_\infty$ . This example serves as a clarification for all the concepts presented in this paper, especially for those readers who are not very familiar with the mathematical tools exploited in our investigation. Let us consider the following process where, for simplicity, the body and the handlers are already presented in terms of  $\mathbf{web}\pi_\infty$ :

$$((z)(\bar{f} \mid z().\mathbf{0}); \mathbf{FAULT} f \rightarrow \overline{\mathit{warning}}; \mathbf{EVENT} e \rightarrow \mathbf{0}; \mathbf{COMP} s \rightarrow \mathbf{0})_s$$

Looking at the previous encoding it results in the following full  $\mathbf{web}\pi_\infty$  process:

$$(e)(f)(y)(y')(y'')(y''')(\langle (z)(\bar{f} \mid z().\mathbf{0}) \mid \bar{y} ; \overline{\mathit{warning}} \mid \bar{y}' \rangle_f \mid !e().\bar{y}'' \mid \langle (x)x() ; s().\bar{y}''' \rangle_y)$$

where  $\mathit{warning}$  is some global channel handling the actual warning (for example displaying a message on the screen). This is a specific instance of the Full Scope Construct as defined above where event and compensation handlers are empty while the fault handler sends an empty message on the warning channel. The process  $z().\mathbf{0}$  expresses the fact that we want the process to fail without allocating the compensation handler and it has to be the standard encoding when raising a failure signal to indicate that there is no successful termination. Now, applying the (FAIL) rule and the floating law, we have:

$$(e)(f)(y)(y')(y'')(y''')(\langle \overline{\mathit{warning}} \mid \bar{y}' ; \mathbf{0} \rangle \mid !e().\bar{y}'' \mid \langle (x)x() ; s().\bar{y}''' \rangle_y)$$

which will lead to a warning on the appropriate channel *without* activating the compensation (which would need a message on  $y$ ) since the scope did not successfully complete. It is worth noting that the event handler remains ready to accept events but it never activates in this scenario. This happens because the channel on which the event handler listens is restricted, and this is consistent with the expected behaviour.

### 5.8 Is It Really Simpler?

The intention of this work is to demonstrate, in real life scenarios, the added value of formal methods. We believe that what has been introduced so far can be really useful in the clarification of the BPEL RF semantic. Just to stress better this point, let us recall only the complete Event Handler compilation presented in [12]:

$$\left( \begin{array}{l} EH(S_e, y_{eh}) = (y')(\{e_x \mid x \in h_e(S_e)\}) \\ \quad \mathit{en}_{eh}().(\langle \prod_{(x, \tilde{u}, A) \in S_e} !x(\tilde{u}).\bar{e}_x \tilde{u} ; \bar{y}_{eh} \rangle_{\mathit{dis}_{eh}} \\ \quad \mid \prod_{(x, \tilde{u}, A_x) \in S_e} !e_x(\tilde{u}).\llbracket A_x \rrbracket_{\bar{y}'}} \end{array} \right)$$

while the new one is:

$$(\mathbf{A}; \mathbf{EVENT} e \rightarrow \mathbf{E})_s \stackrel{\text{def}}{=} (e)(y)(y')(\langle \llbracket A \rrbracket_y ; \mathbf{0} \rangle_y \mid !e().\llbracket E \rrbracket_{y'})$$

For the proper background please refer to [13] where you can find a detailed explanation of the encodings and all the theory. Here the idea is just to give a flavor of how this work contributes (in terms of simplification) to the improvement of the BPEL specification.

### 5.9 Design of BPEL Orchestration Engines

Although this paper has to be intended as investigating a well known case study and providing methodological arguments for the adoption of formal methods in software specification, the aspect of verification is not alien to our work and here we intend to give some hints in this regard. The most common formalization of behavioral equivalence is through barbed congruence, which guarantees that equated processes are indistinguishable by external observers, even when put in arbitrary contexts. For instance, equivalent Web services remain indistinguishable also when composed to form complex business transactions. The barbed congruence in this scenario has been presented in [15]. The proposed encoding, based on the function  $\llbracket A \rrbracket_y : A_{BPEL} \rightarrow Process$ , can be used to test the equivalence of BPEL processes on the basis of the barbed congruence developed in the theory. The idea is to inherit the equivalence notion from  $\mathbf{web}\pi_\infty$  to decide BPEL processes equivalence. Here, as a further contribution, we want to show that, despite its simplicity, there are many ways in which BPEL can benefit from this work exploiting this idea of behavioral equivalence. For example, our proposal can contribute to the implementation of real orchestration engines. The application example comes from one of the theorems proved in [13]:

$$\langle !z(u).P \mid Q \mid \bar{v} \rangle_x \approx_a (y)(\langle !z(u).P \mid \bar{y} \rangle_x \mid \langle Q \mid (w)w(u) \mid \bar{v} \rangle_y)$$

where the symbol  $\approx_a$  has to be intended as barbed congruence, i.e. the process on its left and the one on its right exhibit the same behavior. It is worth noting that the process  $(w)w(u)$  is necessary to prevent  $v$  from disappearing in the case the workunit on the right would terminate successfully. This theorem suggests a transformation where it is always possible to separate the body and the recovery logics of the workunit expressing, for example, the event handler behavior. This is possible not only when the recovery logic is a simple output (as in this case) but in all the other cases, on the basis of another theorem showed in the same work:

$$\langle P \mid Q \rangle_x \approx_a (x')(\langle P \mid \bar{x}' \rangle_x \mid \langle x'().Q \mid \mathbf{0} \rangle)$$

Now we have the design option to compile the mechanism in an alternative way allowing a logical separation of code which can lead to an actual physical separation. For example, different workunits could be loaded on different machines. Although BPEL typically allows a centralized control and a local compilation, this result gives us further insights in the direction of distribution. Consider, for example, the case in which different scopes can share instances of the same handler loaded on a specific dedicated machine. This result can also be interpreted in a choreographic perspective.

## 6 Summary, Related Works and Criticisms

The goal of this paper was to show how a variant of the  $\pi$ -calculus can be of some use in the context of dependable Web services composition. The specific

case study presented aimed at reducing the ambiguity of the BPEL RF providing a (simplified) formal semantics opposed to the complete one already given in [12]. This is what we have called the “ $\pi$ -calculus way”, *i.e.*, using the  $\pi$ -calculus as formal specification language. As we have already underlined, several different formal notations might have been chosen for this purpose. Our choice depended on the “foundational feature” of *mobility*. It has been noted that in the specific contribution the mobility feature has not been fully exploited since the modeled mechanisms required us to pay more attention to process synchronization and concurrency than to full mobility. Anyway, we have realized that, in the general case, mobility is an essential feature of composition languages and this point is discussed more in detail in [13].

Although before this work [15] and [12] have been earlier attempts at defining a formal semantics for WS-BPEL and unifying and simplifying its recovery mechanisms, those papers are far from being complete and from providing the ultimate BPEL formal semantics. Many other works have been presented recently that significantly improved what has been done there. For example, Blite [10] is a “lightweight BPEL” with formal semantics taking into account also dynamic aspects (e.g. dynamic compensations) that have not been directly part of our investigation. Another relevant work adding dynamic compensation features is [20]. In this paper the interested reader can find a comparison between different compensation mechanisms presented in the recent literature. The criticism in this work is that in  $\mathbf{web}\pi_\infty$  completed transactions cannot be compensated. This is of course true but, as shown in this paper, this aspect can be easily modeled (look for example at the encoding of the BPEL compensation handler). The basic idea behind  $\mathbf{web}\pi_\infty$  is indeed to provide a unifying theory for Web services composition as discussed in [15] where different mechanisms can be easily mapped without being directly supported. A good analysis of fault, compensation and termination (FCT) in WS-BPEL is also discussed in [7]. Here the BPEL approach to FCT with related formal semantics is given, thus covering termination handler that has not been part of our work. Furthermore, the authors in [22] recognize that in [12] the lack of support for control links has to be seen as a major drawback. And this is a criticism that we do not hide and we find relevant. The same paper proposes an alternative formalization of WS-BPEL 2.0 based on the  $\pi$ -calculus and then compares different approaches (including the one in [12]) from the complexity point of view for verification purposes. The authors found out that their approach presents a smaller number of states deriving from the neglect of internal activity states. Indeed, while the encoding in [12] requires every activity to signal (at least) its termination (due to the continuation passing style technique used), in [22] the activity lifecycle is not modeled. Apart from the criticisms presented in the recent literature (the list included here is not exhaustive anyway), other interesting questions have been asked regarding this approach to the BPEL RF, for example if we intend to capture fault tolerance behavior depending on external factors, for example timeout. This topic indeed has not been central to our investigation. Other authors worked on these aspects, in particular [9] discusses timed transactions.

Although we know that much needs to be done yet, we are confident that the issues we have identified are worth investigating. We have to admit that sometimes we have doubts regarding what we are doing and the solution we are adopting, so we usually look for some reassurance in the famous words of Descartes: “Dubium Sapientiae initium”, i.e. “Doubt is the origin of wisdom”.

**Acknowledgments.** The paper has been improved during the useful conversations with Cliff Jones, Alexander Romanovsky and Ani Bhattacharyya. For some of the ideas discussed here we have to thank Cosimo Laneve, Roberto Lucchi, Claudio Guidi and Gianluigi Zavattaro. Very useful comments came also by Joey Coleman, Felix Loesch and Michael Jastram that kindly provided other written reviews for this work (Ani Bhattacharyya also provided a written review). Finally, we have also to thank the anonymous WSFM reviewers for their contribution. This work has been partially funded by the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity). More details at <http://www.deploy-project.eu/>.

## References

1. Web services flow language (wsfl 1.0), <http://www.ebpm1.org/wsfl.htm>
2. Xlang: Web services for business process design, <http://www.ebpm1.org/xlang.htm>
3. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (wsdl 1.1), W3C Recommendation (June 26, 2007), <http://www.w3.org/TR/wsdl20/>
4. Chris, P.: Web services orchestration and choreography. *Computer* 36(10), 46–52 (2003)
5. World Wide Web Consortium. Extensible markup language (xml) 1.0. W3C Recommendation: <http://www.w3.org/XML/>
6. Decker, G., Leymann, F., Weske, M.: Bpel4chor: Extending bpel for modeling choreographies. In: *Proceedings International Conference on Web Services, ICWS (2007)*
7. Eisentraut, C., Spieler, D.: Fault, compensation and termination in ws-bpel 2.0 – a comparative analysis. In: Bruni, R., Wolf, K. (eds.) *WS-FM 2008*. LNCS, vol. 5387, pp. 107–126. Springer, Heidelberg (2009)
8. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: Simple object access protocol (soap) 1.1, W3C Recommendation (April 27, 2007), <http://www.w3.org/TR/soap12-part1/>
9. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) *FOSSACS 2005*. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
10. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) *COORDINATION 2008*. LNCS, vol. 5052, pp. 199–215. Springer, Heidelberg (2008)
11. Little, M.: Web services transactions: Past, present and future, <http://www.jboss.org/dms/jbosstm/resources/presentations/XML2003.pdf>
12. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming* 70(1), 96–118 (2007)
13. Mazzara, M.: *Towards Abstractions for Web Services Composition*. PhD thesis, Department of Computer Science, University of Bologna (2006)

14. Mazzara, M., Govoni, S.: A case study of web services orchestration. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 1–16. Springer, Heidelberg (2005)
15. Mazzara, M., Lanese, I.: Towards a unifying theory for web services composition. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 257–272. Springer, Heidelberg (2006)
16. Milner, R.: Functions as processes. *Mathematical Structures in Computer Science* 2(2), 119–141 (1992)
17. Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge (1999)
18. OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language version 2.0., <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>
19. van der Aalst, W.M.P.: Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype (2004), <http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf>
20. Vaz, C., Ferreira, C., Ravara, A.: Dynamic recovering of long running transactions. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 201–215. Springer, Heidelberg (2009)
21. W3C. Http - hypertext transfer protocol, <http://www.w3.org/protocols>
22. Weidlich, M., Decker, G., Weske, M.: Efficient analysis of bpeL 2.0 processes using pi-calculus. In: APSCC 2007: Proceedings of the 2nd IEEE Asia-Pacific Service Computing Conference, Washington, DC, USA, 2007, pp. 266–274. IEEE Computer Society, Los Alamitos (2007)