webπ at work

Cosimo Laneve   Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy.
laneve,zavattar@cs.unibo.it

Abstract. webπ is a recent process calculus that has been inspired by the emerging Web Services technologies. In this paper we explore the expressivity of webπ by discussing two case studies. The first case study is about the formal semantics of the transactional construct of BPEL – the scope construct. The second case study is about a standard pattern of Web Services composition – the *speculative parallelism* – that allows several alternative activities to start; only the first one that completes is taken into account while the other ones are aborted.

1 Introduction

Web Services technologies are emerging mechanisms for describing the services available on the web, as well as their interfaces and the protocols for locating and invoking such services. A challenging issue in this area is the definition of languages and tools for composing services. In fact it is often the case to define new services out of finer-grained subtasks that are likely available as Web Services. As a consequence, several proposals for service composition have been recently devised – the so called *Web Services orchestration and choreography languages.* Among the others we recall XLANG [9], WSFL [8], BPEL [1], and WS-CDL [6].

Long-running transactions are used in most of the Web Services orchestration and choreography languages as basic mechanisms for composing services. These transactions – that we will call *web transactions* – usually do not grant any isolation or atomicity property. As regards isolation, it requires that different activities have the same effect whether they are executed in sequence or in parallel. This is usually enforced by locking the resources used by each activity until the transaction commits. In the context of Web Services, the processes involved in a transaction may belong to different companies, and there is no chance to lock resources of other companies. Additionally, commercial transactions usually last long periods of time, even months, and it is not feasible and not reasonable to block resources so long. For similar reasons, it is not adequate supplying a perfect rollback in the context of Web Services composition. As a matter of facts, in Web Services orchestration and choreography languages, the transaction isolation is delegated to explicit protocols realized through messages; whilst the roll-back mechanisms are defined by ad-hoc programs.

Even if web transactions are attracting a great interest, the Web Services community has not reached a common agreement on a unique notion of this form of transaction. Additionally, the semantics that is usually defined is informal and requires a mathematical analysis. Exceptions we are aware of rely on specific proposals: the work [5] that is mainly inspired by `XLANG`, the calculus of Butler and Ferreira [4] is inspired by `BPBeans`, the πt-calculus [2] considers `BizTalk`, the work [3] deals with short-lived transactions in `BizTalk`.

A different approach has been recently taken in [7], where a process calculus is explicitly designed for modelling web transactions. This calculus, called webπ, is independent of the different proposals and allows to grab the key concepts. Three major aspects are considered in webπ: interruptible processes, failure handlers that are activated when the main process is interrupted, and time. Time has been considered because it is fundamental for dealing with the typical latency of web activities or with message losses. The above three aspects are analyzed in a model consisting of a network of locations that contain processes. In this model time proceeds asynchronously at the network level, while it is constrained by the *local urgency* property inside a location. Local urgency entails the fact that process reductions in a location cannot be delayed to favour idle steps. Said otherwise, local urgency means that the time may elapse in a location either because the process inside progresses or because no progress is possible. We refer to [7] for a discussion about the model of webπ, and its extensional semantics – the *timed bisimilarity*.

The aim of this paper is to explore the expressivity of webπ in two non-trivial case studies inspired by the Web Services technology. The first case study is about the formal semantics of `BPEL`. This language, being the conjoint effort of three major information technology companies, is becoming the standard *de facto* for Web Service orchestration. Defining its formal semantics is therefore a valuable task. In this paper we focus on the unique transactional mechanism in `BPEL` – the `scope` construct – and we define its semantics by means of webπ. The `scope` construct is used to associate a failure handler, a compensation handler, and an event handler to a so-called primary activity. The failure handler is executed if a fault condition occurs during the execution of the primary activity. The compensation handler, on the other hand, is activated if the execution of the primary activity is required to be undone after it has provisionally committed. Finally, the event handler is executed if the primary activity is still under execution and a specific message or allarm is triggered. In this paper we only define the formal semantics of fault and compensation handlers that are used for error handling.

The second case study is about a prototypical pattern of service composition – the so-called *speculative parallelism*. This pattern generalizes the request-response pattern between a client and a server, to cases in which the response can be produced by more than one server. In these cases, the client sends the request to all the possible (alternative) servers. The accepted response is the first one that is received, the other responses are deleted. The non-trivial issue of speculative parallelism is the synchronization of the winner server (the one producing the accepted response) with the communications of failure to the other servers. We model the pattern of speculative parallelism in $\mathtt{web}\pi$, together with a number of erroneous patterns that manifest subtle misbehaviours.

The paper is structured as follows. In Section 2 we recall the syntax ans semantics of $\mathtt{web}\pi$. In Section 3 we discuss the first case study while in Section 4 we discuss the second one. We conclude in Section 5.

2 The calculus $\mathtt{web}\boldsymbol{\pi}$

In this Section we recall the syntax of $\mathtt{web}\pi$ and informally describe its semantics.

The syntax uses a countable set of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$. The syntax of $\mathtt{web}\pi$ includes machines and processes.

$$
\begin{array}{lll}
\mathsf{M} ::= & & \textbf{(machines)} \\
& \mathbf{0} & \text{(nil)} \\
& \mid [\, P \,]_{\widetilde{x}} & \text{(location)} \\
& \mid (x)\mathsf{M} & \text{(machine restriction)} \\
& \mid \mathsf{M} \mid \mathsf{M} & \text{(network)}
\end{array}
\qquad
\begin{array}{lll}
P ::= & & \textbf{(processes)} \\
& \mathbf{0} & \text{(nil)} \\
& \mid \overline{x}\,\widetilde{u} & \text{(message)} \\
& \mid x(\widetilde{u}).P & \text{(input)} \\
& \mid (x)P & \text{(restriction)} \\
& \mid P \mid P & \text{(parallel composition)} \\
& \mid\, !x(\widetilde{u}).P & \text{(replicated input)} \\
& \mid \langle\!| P \;;\; P |\!\rangle^n_x & \text{(transaction)}
\end{array}
$$

A location $[\, P \,]_{\widetilde{x}}$ is a uniprocessor machine; the names $\widetilde{x}$ indicate that the location is responsible for accepting messages on such names. Locations possess their own clock that is not synchronized with the clock of other locations (time progresses asynchronously between different locations). Namely, if M and N are locations, and M evolves in M' then also $\mathsf{M} \mid \mathsf{N}$ evolves in $\mathsf{M}' \mid \mathsf{N}$ (the clock of N remains unchanged).

Processes extend the asynchronous π-calculus with *transactions* $\langle\!| P \;;\; Q |\!\rangle^n_x$, where P and Q are the *body* and the *compensation*, respectively, n indicates the deadline, and x is the name of the transaction. The body of a transaction executes either until termination or until the transaction fails. On failure, the compensation is activated. A transaction may fail in two different ways, either explicitly (when the abort message $\overline{x}$ is consumed, where x is the name of the transaction to be aborted) or implicitly (when the deadline is reached). The deadline may be reached either because of computational steps of the body or because of computational steps of processes in parallel in the same location.

The model of time of $\mathtt{web}\pi$ is such that, within a location, operations cannot be delayed in favour of idle operations – this property is called *local urgency*. For example, consider two processes running on the same location: a printer process of a warning message with a timeout and an idle process waiting for an external event. Local urgency means that, if the external event doesn't occur, then the printer process cannot be delayed. Said otherwise, the time elapses in a location either because the process inside progresses or because no progress is possible.

In $\mathtt{web}\pi$ networks names always index a unique location. The *location names* used in a network are denoted with $\mathtt{ln}(\mathsf{M})$. Formally, let $\mathtt{ln}(\mathsf{M})$ be defined as $\mathtt{ln}(\mathbf{0}) = \emptyset$, $\mathtt{ln}([\, P \,]_{\widetilde{x}}) = \widetilde{x}$, $\mathtt{ln}((x)\mathsf{M}) = \mathtt{ln}(\mathsf{M}) \setminus \{x\}$, and $\mathtt{ln}(\mathsf{M} \mid \mathsf{N}) = \mathtt{ln}(\mathsf{M}) \cup$

$\mathtt{ln}(\mathsf{N})$. Networks $\mathsf{M} \mid \mathsf{N}$ are constrained to satisfy the property $\mathtt{ln}(\mathsf{M}) \cap \mathtt{ln}(\mathsf{N}) = \emptyset$. This constraint permits to deliver messages to the unique machine in the network that is responsible for accepting that message. However it is also possible to consume messages in the same machine in which they have been produced as it will be made clear by the formal definition below.

The input $x(\widetilde{u}).P$, restriction $(x)P$, replicated input $!x(\widetilde{u}).P$, and machine restriction $(x)\mathsf{M}$ are binders of names $\widetilde{u}$, x, and $\widetilde{u}$, and x, respectively. The scope of these binders are the processes P and the machine M. We use the standard notions of α-equivalence, *free* and *bound names* of processes, and machines, noted $\mathtt{fn}(P)$, $\mathtt{bn}(P)$, $\mathtt{fn}(\mathsf{M})$, $\mathtt{bn}(\mathsf{M})$, respectively. In particular,

- $\mathtt{fn}(\langle\!| P \,;\, Q |\!\rangle_x^n) = \mathtt{fn}(P) \cup \mathtt{fn}(Q) \cup \{x\}$ and α-equivalence equates $(x)(\langle\!| P \,;\, Q |\!\rangle_x^n)$ with $(z)(\langle\!| P\{^z/_x\} \,;\, Q\{^z/_x\} |\!\rangle_z^n)$ provided $z \notin \mathtt{fn}(\langle\!| P \,;\, Q |\!\rangle_x^n)$;
- $\mathtt{fn}([\, P \,]_{\widetilde{x}}) = \widetilde{x} \cup \mathtt{fn}(P)$.

Following the tradition of π-calculus, the semantics of $\mathtt{web}\pi$ is defined in terms of a reduction relations that relies on a structural congruence that equates all terms one never wants to distinguish.

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

for processes:

1. *the scope laws:*

$$\begin{array}{c}(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P, \\ P \mid (u)Q \equiv (u)(P \mid Q)\ , \quad \textit{if } u \notin \mathtt{fn}(P) \\ \langle\!| (z)P \,;\, Q |\!\rangle_x^n \equiv (z)\langle\!| P \,;\, Q |\!\rangle_x^n\ , \quad \textit{if } z \notin \{x\} \cup \mathtt{fn}(Q) \\ \langle\!| P \,;\, (z)Q |\!\rangle_x^0 \equiv (z)\langle\!| P \,;\, Q |\!\rangle_x^0\ , \quad \textit{if } z \notin \{x\} \cup \mathtt{fn}(P)\end{array}$$

2. *the repetition law:*

$$!x(\widetilde{u}).P \equiv x(\widetilde{u}).P \mid !x(\widetilde{u}).P$$

3. *the transaction laws:*

$$\begin{array}{c}\langle\!| \mathbf{0} \,;\, Q |\!\rangle_x^n \equiv \mathbf{0} \\ \langle\!| \langle\!| P \,;\, Q |\!\rangle_y^n \mid R \,;\, R' |\!\rangle_x^m \equiv \langle\!| P \,;\, Q |\!\rangle_y^n \mid \langle\!| R \,;\, R' |\!\rangle_x^m\end{array}$$

4. *the floating laws:*

$$\begin{array}{c}\langle\!| \overline{z}\,\widetilde{u} \mid P \,;\, Q |\!\rangle_x^n \equiv \overline{z}\,\widetilde{u} \mid \langle\!| P \,;\, Q |\!\rangle_x^n \\ \langle\!| y(\widetilde{v}).P \mid P' \,;\, \overline{z}\,\widetilde{u} \mid Q |\!\rangle_x^0 \equiv \overline{z}\,\widetilde{u} \mid \langle\!| y(\widetilde{v}).P \mid P' \,;\, Q |\!\rangle_x^0\end{array}$$

for machines:

1. *the machine scope laws:*

$$\begin{array}{c}(u)\mathbf{0} \equiv \mathbf{0}, \qquad (x)(z)\mathsf{M} \equiv (z)(x)\mathsf{M}, \\ \mathsf{M} \mid (x)\mathsf{N} \equiv (x)(\mathsf{M} \mid \mathsf{N})\ , \quad \textit{if } x \notin \mathtt{fn}(\mathsf{M}) \\ [\, (x)P \,]_{\widetilde{z}} \equiv (x)[\, P \,]_{\widetilde{z}x}\ , \quad \textit{if } x \notin \widetilde{z}\end{array}$$

2. the lifting law:

$$[\,P\,]_{\widetilde{x}} \equiv [\,Q\,]_{\widetilde{x}}\,, \qquad \text{if } P \equiv Q$$

The dynamic behaviour of processes and machines is defined by the reduction relation. The operation of decreasing by 1 the time stamps of active transactions on the same machine is modelled by the *time stepper function*. The definitions of this function and another auxiliary function are in order:

input predicate $\mathtt{inp}(P)$: this predicate verifies whether a process contains an input that is not underneath a transaction. It is the least relations such that:

$$\begin{array}{ll} \mathtt{inp}(x(\widetilde{u}).P) & \\ \mathtt{inp}((x)P) & \text{if } \mathtt{inp}(P) \\ \mathtt{inp}(P \mid Q) & \text{if } \mathtt{inp}(P) \text{ or } \mathtt{inp}(Q) \\ \mathtt{inp}(!x(\widetilde{u}).P) & \end{array}$$

time stepper function $\phi(P)$: this function decreases the time stamps by 1. For the missing cases, $\phi(P) = P$.

$$\begin{array}{rcl} \phi((x)P) & = & (x)\phi(P) \\ \phi(P \mid Q) & = & \phi(P) \mid \phi(Q) \\ \phi(\langle\!| P \; ; \; R |\!\rangle_x^0) & = & \begin{cases} \langle\!| \phi(P) \; ; \; \phi(R) |\!\rangle_x^0 & \text{if } \mathtt{inp}(P) \\ \langle\!| \phi(P) \; ; \; R |\!\rangle_x^0 & \text{otherwise} \end{cases} \\ \phi(\langle\!| P \; ; \; R |\!\rangle_x^{n+1}) & = & \langle\!| \phi(P) \; ; \; R |\!\rangle_x^n \end{array}$$

We are finally in place for defining the reduction relation.

Definition 2. *The* reduction relation $\rightarrow$ *is the least relation satisfying the reductions:*

for processes:

$$\overset{(\text{COM})}{\overline{x}\,\widetilde{v} \mid x(\widetilde{u}).P \;\rightarrow\; P\{\widetilde{v}/\widetilde{u}\}}$$

$$\overset{(\text{FAIL})}{\overline{x} \mid \langle\!| z(\widetilde{u}).P \mid Q \; ; \; R |\!\rangle_x^{n+1} \;\rightarrow\; \langle\!| z(\widetilde{u}).P \mid \phi(Q) \; ; \; R |\!\rangle_x^0}$$

and closed under $\equiv$, (x)-, *and the rules:*

$$\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid \phi(R)} \qquad \frac{P \rightarrow Q}{\langle\!| P \; ; \; R |\!\rangle_x^{n+1} \rightarrow \langle\!| Q \; ; \; R |\!\rangle_x^n} \qquad \frac{P \rightarrow Q}{\langle\!| y(\widetilde{v}).R \mid R' \; ; \; P |\!\rangle_x^0 \rightarrow \langle\!| y(\widetilde{v}).R \mid \phi(R') \; ; \; Q |\!\rangle_x^0}$$

for machines:

$$\overset{(\text{INTRA})}{\frac{P \rightarrow Q}{[\,P\,]_{\widetilde{x}} \rightarrow [\,Q\,]_{\widetilde{x}}}} \qquad \overset{(\text{TIME})}{\frac{P \not\rightarrow}{[\,P\,]_{\widetilde{x}} \rightarrow [\,\phi(P)\,]_{\widetilde{x}}}} \qquad \overset{(\text{DELIV})}{[\,\overline{x}\,\widetilde{v} \mid P\,]_{\widetilde{z}} \mid [\,Q\,]_{\widetilde{y}x} \rightarrow [\,P\,]_{\widetilde{z}} \mid [\,\overline{x}\,\widetilde{v} \mid Q\,]_{\widetilde{y}x}}$$

and closed under $\equiv$, (x)-, *and parallel composition.*

It is worth noting that a message can be either consumed in the machine where it has been produced, see rule (COM), or in the machine responsible for managing the channel on which the message was sent, see rule (DELIV).

We illustrate the semantics by discussing few examples. The process

$$\overline{z} \mid \overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^n$$

has the following two computations ($n > 0$):

$$\overline{z} \mid \overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^n \ \rightarrow \ \overline{z} \mid \langle\!| \mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^{n-1}$$

$$\overline{z} \mid \overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^n \ \rightarrow \ \overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^0$$

Transactions with time stamps equal to 0 are *terminated.* There are two kinds of terminated transactions: (a) the *committed transactions*, such as $\langle\!| \mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^{n-1}$ that is (structurally) equivalent to $\langle\!| \mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^0$, whose bodies do not contain input-guarded processes, and (b) the *failed transactions*, such as $\langle\!| x().\mathbf{0} \ ; \ \overline{y} \,|\!\rangle_z^0$, whose body contains input-guarded processes. The input operations in the body of failed transactions can no longer be executed: the transaction is actually failed because it has not completed its tasks. In committed transactions the compensation handler is no more considered. This is reflected in the first computation by the fact that the message $\overline{y}$ cannot be produced. This message is syntactically part of the process but it cannot be consumed. In failed transactions the compensation process becomes active. This is made clear in the second computation, where the message $\overline{z}$ explicitly abort the transaction thus making the time stamp equal to 0. After abortion, the message $\overline{x}$ cannot be consumed.

The process $\overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{u} \,|\!\rangle_z^1 \mid \langle\!| x().\mathbf{0} \ ; \ \overline{v} \,|\!\rangle_{z'}^1$ evolves as follows

$$\overline{x} \mid \langle\!| x().\mathbf{0} \ ; \ \overline{u} \,|\!\rangle_z^1 \mid \langle\!| x().\mathbf{0} \ ; \ \overline{v} \,|\!\rangle_{z'}^1 \quad \rightarrow \quad \langle\!| \mathbf{0} \ ; \ \overline{u} \,|\!\rangle_z^0 \mid \langle\!| x().\mathbf{0} \ ; \ \overline{v} \,|\!\rangle_{z'}^0$$

(and in a similar way, but consuming the input of z' instead of the input of z). This reduction shows the progress of time in a location: a computational step of a process makes the time elapse of one unit. This is manifested by decreasing the time stamps of every transaction in parallel (in the previous case, of $\langle\!| x().\mathbf{0} \ ; \ \overline{v} \,|\!\rangle_{z'}^1$). We note that the transaction $\langle\!| \mathbf{0} \ ; \ \overline{u} \,|\!\rangle_z^0$ is completed, therefore the message $\overline{u}$ is never emitted. On the contrary, $\langle\!| x().\mathbf{0} \ ; \ \overline{v} \,|\!\rangle_{z'}^0$ is failed, thus $\overline{v}$ is emitted.

In `web`π, the delivery of one message to its receptor machine is modeled by the computation step $[\, \overline{x}\widetilde{w} \mid P \,]_{\widetilde{y}} \mid [\, Q \,]_{\widetilde{z}x} \ \rightarrow \ [\, P \,]_{\widetilde{y}} \mid [\, \overline{x}\widetilde{w} \mid Q \,]_{\widetilde{z}x}$. Asynchrony between machines may give rise to unpredictable delays in communication. This latency could make timed transaction fail. Consider, for instance, the machine (trailing $\mathbf{0}$ are omitted)

$$[\, \langle\!| \overline{x} \mid y() \ ; \ \overline{z} \,|\!\rangle_x^n \,]_y \ \mid \ [\, x().\overline{y} \,]_x$$

where the leftmost location sends the message $\overline{x}$ to the rightmost one and waits for the answer $\overline{y}$. Due to asynchrony between machines, the following computation is possible (let $m < n$ and $n < m + m'$):

$$
\begin{array}{rl}
[\![\, \langle\!| \overline{x} \mid y() \,;\, \overline{z} \,|\!\rangle_x^n \,]\!]_y \mid [\![\, x().\overline{y} \,]\!]_x & \rightarrow^m [\![\, \overline{x} \mid \langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^{n-m} \,]\!]_y \mid [\![\, x().\overline{y} \,]\!]_x \\
& \rightarrow [\![\, \langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^{n-m} \,]\!]_y \mid [\![\, \overline{x} \mid x().\overline{y} \,]\!]_x \\
& \rightarrow [\![\, \langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^{n-m} \,]\!]_y \mid [\![\, \overline{y} \,]\!]_x \\
& \rightarrow^{m'} [\![\, \langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^{0} \,]\!]_y \mid [\![\, \overline{y} \,]\!]_x \\
& \rightarrow [\![\, \overline{y} \mid \langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^{0} \,]\!]_y \mid [\![\, \mathbf{0} \,]\!]_x
\end{array}
$$

where $\rightarrow^k$ is used to denote the effect of k subsequent reductions. In the final state the message $\overline{y}$ cannot be consumed by the transaction $\langle\!| y() \,;\, \overline{z} \,|\!\rangle_x^0$ as the time stamp is 0.

A usual source of failure in networks is the loss of messages. Such failures have not been explicitly considered in `web`π because they are modelled by indefinitely delaying messages.

3 The scope construct in `BPEL`

The first case study we discuss is the modelling of the the `scope` construct of `BPEL`. This construct defines transactional activities by associating a failure handler and a compensation handler to a primary activity. The `scope` construct in `BPEL` also specifies an event handler. In our simplified modeling we only consider fault and compensation handlers used to deal with exceptional bahaviours.

The failure handler is activated in case a fault condition occurs during the execution of the primary activity. The compensation handler is executed in case the execution of the primary activity is required to be undone after is has provisionally committed. In fact, the primary activity could be part of a more complex task that fails, thus requiring to cancel those subactivities that provisionally completed.

In this section we consider only the timeless transactions of `web`π as we are mainly interested in the flow of control in the execution of fault and compensation handlers, independently of the time constraints that can be associated to the execution of the primary activity within a `BPEL` `scope`. Indeed, a timeout can be associated to a `scope` using an event handler (the third form of handler that we do not model) that is triggered by a timed allarm.

We denote the `scope` construct with $\mathtt{scope}_x(P \,;\, F \,;\, C)$, where P is the primary activity, F is the failure handler, and C is the compensation handler. The name x is used to signal either the occurrence of a failure during the execution of the primary activity, or the external request of compensation. We assume that P, F, and C are asynchronous π-calculus processes.

Before discussing the `web`π semantics of `scope`, we present a prototypal example about scopes. Consider a travel organization service that requires to orchestrate an hotel and a flight reservation service. We first define the last two services, then we show how to orchestrate them using the `scope` construct.

The hotel reservation service is (abstractly) modeled as follows:

$$\begin{array}{rl}\mathsf{HOTEL} = [& !res_h(arr, dep, conf_h).(id_h)(\overline{conf_h}\langle id_h\rangle \mid \overline{id_h}) \\ & \mid\ !cancel_h(id_h).id_h() \\ ]_{res_h, cancel_h} & \end{array}$$

The service receives reservation requests indicating the arrival date *arr*, the departure date *dep*, and a channel to be used for communicating the reservation confirmation $conf_h$. Each reservation has a unique identifier id_h, which is communicated through the reservation confirmation channel $conf_h$. The service keeps track of the reservation by producing an internal message $\overline{id_h}$. In case of cancellation, this message is consumed. This occurs when the name id_h is received back through the channel $cancel_h$.

The flight registration service is modeled similarly, with the unique difference that the reservation could fail. Let $P \oplus Q$ be the process $(x)(\overline{x} \mid x().P \mid x().Q)$, assuming that $x \notin \mathtt{fv}(P) \cup \mathtt{fv}(Q)$.

$$\begin{array}{rl}\mathsf{FLIGHT} = [& !res_f(arr, dep, conf_h, t).\quad ((id_f)(\overline{conf_f}\langle id_f\rangle \mid \overline{id_f}) \oplus \overline{t}) \\ & \mid\ !cancel_f(id_f).id_f() \\ ]_{res_f, cancel_f} & \end{array}$$

The flight reservation request, besides the arrival and departure dates and the confirmation channel, carries a fourth name t used in case of reservation failure. After the reservation request is received, the service internally choose either to accept or reject it; in case of failure, a message on the channel t is produced.

We are now in place for programming an orchestrator – the travel organization service – that combines the above services:

$$\begin{array}{rl}\mathsf{TRVL} = & (conf_h, conf_f, store_h, store_f, t) \\ & [\quad \mathtt{scope}_t(\ \overline{res_h}\langle arr, dep, conf_h\rangle \\ & \qquad\qquad \mid conf_h(id_h).(\ \overline{store_h}\langle id_h\rangle \mid \overline{res_f}\langle arr, dep, conf_f, t\rangle \\ & \qquad\qquad\qquad \mid conf_f(id_f).\overline{store_f}\langle id_f\rangle\ ) \\ & \quad ;\ store_h(id_h).\overline{cancel_h}\langle id_h\rangle \\ & \quad ;\ store_h(id_h).\overline{cancel_h}\langle id_h\rangle \mid store_f(id_f).\overline{cancel_f}\langle id_f\rangle\quad) \\ &]_{conf_h, conf_f, t}\end{array}$$

The travel service uses two local channels $store_h$ and $store_f$ to store the identifiers of the hotel and flight reservations, respectively. The *primary activity* first sends a request to the hotel reservation service, then to the flight reservation service. The *failure* handler manages those cases in which the flight reservation does not succeed; in these cases the hotel reservation is cancelled. The *compensation* handler, on the other hand, manages those cases (that we do not model explicitly) in which the travellers decides to cancel its travel after it has been fully reserved; in this case both the hotel and the flight reservations are cancelled.

The whole reservation system is modeled as the parallel composition of the three services described above:

$$\mathsf{HOTEL} \mid \mathsf{FLIGHT} \mid \mathsf{TRVL}$$

The semantics of $\mathtt{scope}$ is defined by the following function $[\![\cdot]\!]$ translating the term $\mathtt{scope}_x(P\ ;\ F\ ;\ C)$ for any name x and asynchronous π-calculus processes P, F, and C. Let $z_f, z_c \notin \{x\} \cup \mathtt{fn}(P \,|\, F \,|\, C)$.

$$[\![\mathtt{scope}_x(P\ ;\ F\ ;\ C)]\!] = (z_c, z_f) \\ \langle\!|\, \overline{z_f} \,|\, (y)([\![P]\!]_y \,|\, y().z_f().(\overline{z_c} \,|\, (v)v())) \ ;\ z_f().F \,|\, z_c().C \,|\!\rangle_x$$

The $\mathtt{web}\pi$ process associated to $\mathtt{scope}$ is a timeless transaction having the same name x. The body of this transaction cannot commit because of the ending process $(v)v()$ that is deadlocked. The channels z_f and z_c are used to indicate whether the failure or the compensation handler should be activated in case the transaction is aborted. In particular, the message $\overline{z_f}$ activates the failure handler, while $\overline{z_c}$ activates the compensation handler. The message $\overline{z_f}$ is present during the execution of the primary activity. If the primary activity completes, the message $\overline{z_f}$ is replaced by $\overline{z_c}$. In order to detect the completion of the primary activity P we use a continuation passing style.

Let $[\![P]\!]_y$ be the function that executes P and produces $\overline{y}$ when P terminates. Let also assume that y, y', y'' are always fresh names:

$$\begin{array}{rcl} [\![\mathbf{0}]\!]_y & = & \overline{y} \\ [\![\overline{x}\,\widetilde{u}]\!]_y & = & \overline{x}\,\widetilde{u} \,|\, \overline{y} \\ [\![x(\widetilde{u}).P]\!]_y & = & x(\widetilde{u}).[\![P]\!]_y \\ [\![(x)P]\!]_y & = & (x)[\![P]\!]_y \\ [\![P \,|\, Q]\!]_y & = & (y', y'')([\![P]\!]_{y'} \,|\, [\![Q]\!]_{y''} \,|\, y'().y''().\overline{y}\,) \\ [\![!x(\widetilde{u}).P]\!]_y & = & !x(\widetilde{u}).[\![P]\!]_y \end{array}$$

The definition of $[\![P]\!]_y$ is standard; we comment only the rule dealing with the parallel composition $P \,|\, Q$. Two new names y' and y'' are used to communicate the completion of the two processes P and Q, respectively. When both $\overline{y'}$ and $\overline{y''}$ are produced, the overall process completes (thus $\overline{y}$ is produced).

4 Speculative parallelism

The second case study is about a special pattern of client-services interaction: the so-called *speculative parallelism*. Speculative parallelism is used by a client that engages (in parallel) request-response interactions with several services in such a way that if one of the services completes – the *winner* –, the remaining services – the *losers* – are abandoned.

Before discussing a formal representation of speculative parallelism, we consider a simpler case of request-response protocol between one client and one service. The protocol is modelled in $\mathtt{web}\pi$ using the following network:

$$\begin{array}{rl} \mathsf{RP} = & [\ \langle\!|\, \overline{req}\,.resp().(\overline{ack} \,|\, \overline{success}\,)\ ;\ \overline{fail} \,|\, \overline{nack}\,|\!\rangle_t^n\ ]_{resp} \\ | & [\ req().(\overline{resp} \,|\, ack().\overline{done} \,|\, nack().\overline{abort}\,)\ ]_{req,ack,nack} \end{array}$$

The client (the machine on the first line) sends a request $\overline{req}$ to the service (the machine on the second line) and blocks waiting for the response message $\overline{resp}$.

If the response arrives in due time (i.e. before the timeout n expires), the client produces the message $\overline{success}$; otherwise, it produces the message $\overline{fail}$. The service produces the message $\overline{done}$ in the case the request-response interaction succeeds, $\overline{abort}$ otherwise. This is achieved by letting the client to produce $\overline{ack}$ (respectively, $\overline{nack}$) when it succeeds (respectively, fails).

Informally, the request-response protocol is correct if it satisfies the following property: *every partial computation may be completed in such a way that both the client and the service communicate their final state; moreover, the final states of the client and the service are consistent.* The transliteration of this property in the network RP is: "every computation may be completed in such a way that the client emits $\overline{success}$ or $\overline{fail}$, while the service emits $\overline{done}$ or $\overline{abort}$; additionally, $\overline{success}$ and $\overline{abort}$ cannot be both produced, as well as $\overline{fail}$ and $\overline{done}$".

To be more formal, let $\mathsf{M} \downarrow x$, read M *has barb* x, be the predicate defined by

$$\mathsf{M} \downarrow x \quad \text{if and only if} \quad \mathsf{M} \equiv (\widetilde{y})(\,[\,\overline{x}\,\widetilde{w} \mid P\,]_{\widetilde{z}} \mid \mathsf{N}) \text{ for some } \widetilde{y}, \widetilde{w}, P, \widetilde{z}, \mathsf{N}$$

The following auxiliary notations are also used:

$$\begin{array}{ll} \mathsf{M} \downarrow \langle x_1 \dots x_n \rangle & \text{if } \mathsf{M} \downarrow x_i \text{ for } i \in 1 \dots n \\ \mathsf{M} \not\downarrow \langle x_1 \dots x_n \rangle & \text{if } \mathsf{M} \downarrow \langle x_1 \dots x_n \rangle \text{ does not hold} \\ \mathsf{M} \Downarrow \langle x_1 \dots x_n \rangle & \text{if } \mathsf{M} \to^* \mathsf{M}' \text{ for some } \mathsf{M}' \text{ and } \mathsf{M}' \downarrow \langle x_1 \dots x_n \rangle \end{array}$$

where $\to^*$ denotes the reflexive and transitive closure of the reduction relation $\to$ defined for machines. Then the correctness property may be rewritten as follows: for every machine M such that $\mathsf{RP} \to^* \mathsf{M}$, the following two conditions hold:

- $\mathsf{M} \Downarrow \langle success, done \rangle$ or $\mathsf{M} \Downarrow \langle fail, abort \rangle$,
- $\mathsf{M} \not\downarrow \langle fail, done \rangle$ and $\mathsf{M} \not\downarrow \langle success, abort \rangle$.

It is not difficult to verify that RP is a correct request-response protocol.

We now move to the more general case of speculative parallelism. For simplicity, we consider the case of one client and two services; the generalization to more than two services is trivial. Let the client send in parallel two requests to two different services. If at least one response reaches the client in due time, that service is completed and the other one must be aborted. If no response arrives before the time-out expires, both services must be aborted.

The first machine we discuss is a direct adaptation of RP:

$$\begin{array}{rl} \mathsf{SP1} \;=& [\,(f)(\; f().\overline{fail} \\ & \qquad \mid \langle\!|\, \overline{req1} \mid resp1().(\overline{ack1} \mid \overline{success} \mid \overline{t2}\,) \;;\; \overline{f} \mid \overline{nack1}\,|\!\rangle^n_{t1} \\ & \qquad \mid \langle\!|\, \overline{req2} \mid resp2().(\overline{ack2} \mid \overline{success} \mid \overline{t1}\,) \;;\; \overline{f} \mid \overline{nack2}\,|\!\rangle^n_{t2}\;) \\ &]_{resp1, resp2} \\ & \mid \quad [\,req1().(\overline{resp1} \mid ack1().\overline{done1} \mid nack1().\overline{abort1}\,)\,]_{req1, ack1, nack1} \\ & \mid \quad [\,req2().(\overline{resp2} \mid ack2().\overline{done2} \mid nack2().\overline{abort2}\,)\,]_{req2, ack2, nack2} \end{array}$$

The locations in the last two lines are the two services. They behave in much the same way as the service in RP (the difference is that we use the indexes 1 and 2 to separate them). The client performs two transactions similar to the one performed by the client in RP. Each transaction engages an interaction with the corresponding service. The difference with RP is that, in case of success of one transaction, the other transaction is aborted explicitly (using the message $\overline{t1}$ or $\overline{t2}$). The local name f is used to implement failure. This is necessary in order to avoid that two instances of $\overline{fail}$ are produced when, e.g., the time-out expires. In this last case both the transactions fail and the two compensations are activated.

To analyse the correctness of $\mathsf{SP1}$, we generalize to two services the above property. Let $\mathsf{SP1}$ be correct if the following property holds: *for every machine* M *such that* $\mathsf{SP1} \to^* \mathsf{M}$*, the following two conditions hold:*

– $\mathsf{M} \Downarrow \langle success, done1, abort2\rangle$ *or* $\mathsf{M} \Downarrow \langle success, abort1, done2\rangle$ *or* $\mathsf{M} \Downarrow \langle fail, abort1, abort2\rangle$,
– $\mathsf{M} \not\Downarrow \langle done1, done2\rangle$ *and* $\mathsf{M} \not\Downarrow \langle fail, done1, abort2\rangle$ *and* $\mathsf{M} \not\Downarrow \langle fail, abort1, done2\rangle$ *and* $\mathsf{M} \not\Downarrow \langle success, abort1, abort2\rangle$.

We notice that $\mathsf{SP1}$ is incorrect because it may happen that both transactions commit. This occurs if both messages $\overline{resp1}$ and $\overline{resp2}$ reach the client when the time stamp is $n' > 1$. For instance, consider the computation

$$
\begin{array}{l}
\mathsf{SP1} \;\to^*\; [\!\![\quad \overline{resp1} \mid \overline{resp2} \\
\qquad\qquad \mid (f)(\; f().\overline{fail} \\
\qquad\qquad\qquad \mid \langle\!| resp1().(\overline{ack1} \mid \overline{success} \mid \overline{t2}\,) \;;\; \overline{f} \mid \overline{nack1} \,|\!\rangle_{t1}^{n'} \\
\qquad\qquad\qquad \mid \langle\!| resp2().(\overline{ack2} \mid \overline{success} \mid \overline{t1}\,) \;;\; \overline{f} \mid \overline{nack2} \,|\!\rangle_{t2}^{n'} \;) \\
\qquad]\!\!]_{resp1,resp2} \\
\qquad \mid \quad [\!\![\, ack1().\overline{done1} \mid nack1().\overline{abort1} \,]\!\!]_{req1,ack1,nack1} \\
\qquad \mid \quad [\!\![\, ack2().\overline{done2} \mid nack2().\overline{abort2} \,]\!\!]_{req2,ack2,nack2}
\end{array}
$$

It is easy to verify that this computation may be completed yielding a machine M such that $\mathsf{M} \downarrow \langle done1, done2\rangle$. This contradicts the second condition of the previous property.

This problem may be avoided by enclosing the two transactions in an outermost transaction that is responsible to check that at most one transaction succeeds. This solution is implemented by the machine $\mathsf{SP2}$.

$$
\begin{array}{l}
\mathsf{SP2} \;=\; [\!\![\, (r, a1, a2)\, \langle\!| \quad \langle\!| \overline{req1} \mid resp1().(\overline{r}\,\langle t2, a1\rangle \mid a1().\overline{ack1}\,) \;;\; \overline{nack1} \,|\!\rangle_{t1} \\
\qquad\qquad\qquad\qquad \mid \langle\!| \overline{req2} \mid resp2().(\overline{r}\,\langle t1, a2\rangle \mid a2().\overline{ack2}\,) \;;\; \overline{nack2} \,|\!\rangle_{t2} \\
\qquad\qquad\qquad\qquad \mid r(u, v).(\overline{u} \mid \overline{v} \mid \overline{success}\,) \\
\qquad\qquad\qquad\qquad ;\; \overline{t1} \mid \overline{t2} \mid \overline{fail} \quad |\!\rangle_{t}^{n} \\
\qquad]\!\!]_{resp1,resp2} \\
\qquad \mid \quad [\!\![\, req1().(\overline{resp1} \mid ack1().\overline{done1} \mid nack1().\overline{abort1}\,) \,]\!\!]_{req1,ack1,nack1} \\
\qquad \mid \quad [\!\![\, req2().(\overline{resp2} \mid ack2().\overline{done2} \mid nack2().\overline{abort2}\,) \,]\!\!]_{req2,ack2,nack2}
\end{array}
$$

This machine is correct. The formal proof of this result is not reported, as it is a tedious analysis of the possible computations. We report an informal discussion of the basic idea underlying the implementation.

The request-response interactions with the services are realized by the transactions $t1$ and $t2$, which are inside a transaction t that is responsible for deciding the winner and the loser. The transactions $t1$ and $t2$ send a request to the corresponding service and wait for the answer. On reception of the answer, $t1$ and $t2$ communicate their end on the private channel r. The message carries two names: the first one is the name of the opposite transaction while the second one is the name of an input where the transaction body is waiting for an acknowledgement. When the t-transaction receives these two names, they are used to cancel the loser and to acknowledge the winner. Both $t1$ and $t2$ have an associated compensation process that may cancel the task itself. The compensation process of t simply invokes the compensations of $t1$ and $t2$.

As regards time, the time stamp n is associated to the transaction t, while $t1$ and $t2$ are timeless. It is worth noting that, if $t1$ and $t2$ where timed, the protocol turns out to be incorrect. In fact, the time-outs of $t1$ and $t2$ may expire after the transaction t has received a message on the channel r, but before the winner transaction is notified. To clarify this circumstance, let $\mathsf{SP2}'$ be the machine $\mathsf{SP2}$ where the time stamp n is also associated to the transactions $t1$ and $t2$. It is possible to obtain

$$
\begin{array}{lll}
\mathsf{SP2}' \to^* & [\,(r, a1, a2)\,\langle\!| & \langle\!| a1().\overline{ack1} \;;\; \overline{nack1}\,|\!\rangle^{0}_{t1} \\
& & \mid\; \langle\!| \overline{req2} \mid resp2().(\overline{r}\,\langle t1, a2\rangle \mid a2().\overline{ack2}\,) \;;\; \overline{nack2}\,|\!\rangle^{0}_{t2} \\
& & \mid\; \overline{t2} \mid \overline{a1} \mid \overline{success} \\
& & ;\; \overline{t1} \mid \overline{t2} \mid \overline{fail} \quad |\!\rangle^{0}_{t} \\
&]_{resp1,resp2} & \\
& \mid \quad [\,ack1().\overline{done1} \mid nack1().\overline{abort1}\,]_{req1,ack1,nack1} & \\
& \mid \quad [\,req2().(\overline{resp2} \mid ack2().\overline{done2} \mid nack2().\overline{abort2}\,)\,]_{req2,ack2,nack2} &
\end{array}
$$

The reader may verify that this computation may be extended reaching a machine M such that $\mathsf{M} \downarrow \langle success, abort1, abort2\rangle$ thus contradicting the second condition of the property formalized above.

5 Conclusion

We have explored the expressivity of `webπ` for modeling and reasoning about typical mechanisms of Web Services orchestration and composition. In particular, two case studies have been considered, one inspired by the orchestration language `BPEL` and another one based on the pattern of services combination known as speculative parallism.

In the next future we intend to consider a more significant fragment of `BPEL`, in particular the so-called event handlers, as well as composition mechanisms of other emerging languages such as WS-CDL. We also intend to model and compare a whole class of patterns of composition for services. In this respect, the library of patterns described in [10] will be taken as the main source of inspiration for the protocols to be considered.

References

1. T. Andrews and et.al. Business process execution language for web services. Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, 2003.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long running transactions. In *FMOODS'03, Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 124–138. Springer-Verlag, 2003.
3. R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. In *CONCUR 2002: Proceedings of the 13th International Conference on Concurrency Theory*, volume 2421 of *LNCS*, pages 321–337. Springer Verlag, 2002.
4. M. Butler and C. Ferreira. An operational semantics for stac, a language for modelling long-running business transactions. In *COORDINATION'04, Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 87–104. Springer-Verlag, 2004.
5. M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In Proceedings of 25 Years of CSP, London, 2004.
6. N. Kavantzas, G. Olsson, J. Mischkinsky, and M. Chapman. Web services choreography description languages. W3C Web Services Choreography Working Group, 2003.
7. C. Laneve and G. Zavattaro. Foundations of web transactions. In *FOSSACS 2005: Proceedings of Foundations of Software Science and Computation Structure*, volume to appear of *LNCS*. Springer Verlag, 2005.
8. F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM Software Group, 2001.
9. S. Thatte. XLANG: Web services for business process design. Microsoft Corporation, 2001.
10. M. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.