A Case Study of Web Services Orchestration

Manuel Mazzara[1] and Sergio Govoni[2]

[1] Department of Computer Science, University of Bologna, Italy
mazzara@cs.unibo.it
[2] Imaging Science and Information Systems Center, Georgetown University, Washington, DC, USA
govoni@isis.imac.georgetown.edu

Abstract. Recently the term Web Services Orchestration has been introduced to address composition and coordination of Web Services. Several languages to describe orchestration for business processes have been presented and many of them use concepts such as long-running transactions and compensations to cope with error handling. WS-BPEL is currently the best suited in this field. However, its complexity hinders rigorous treatment. In this paper we address the notion of orchestration from a formal point of view, with particular attention to transactions and compensations. In particular, we discuss $\texttt{web}\pi_\infty$, an untimed subcalculus of $\texttt{web}\pi$ [15] which is a simple and conservative extension of the π-calculus. We introduce it as a theoretical and foundational model for Web Services coordination. We simplify some semantical and pragmatical aspects, in particular regarding temporization, gaining a better understanding of the fundamental issues. To discuss the usefulness of the language we consider a case study: we formalize an e-commerce transactional scenario drawing on a case presented in our previous work [12].

1 Introduction

The aim of Web Services is to ease and to automate business process collaborations across enterprise boundaries. The core Web Services standards, WSDL [11] and UDDI [26], cover calling services over the Internet and finding them, but they are not enough. Creating collaborative processes requires an additional layer on top of the Web Services protocol stack: this way we can achieve Web Services composition and orchestration. In particular, orchestration is the description of interactions and messages flow between services in the context of a business process [23]. Orchestration is not a new concept; in the past it has been called workflow [28].

1.1 The State of the Art in Orchestration

Three specifications have been introduced to cover orchestration: Web Services Business Process Execution Language (WS-BPEL or BPEL for short) [1] which is the successor of Microsoft XLANG [25, 5] and IBM WSFL [16], together

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 1–16, 2005.

with WS-Coordination (WS-C) [29] and WS-Transaction (WS-T) [30]. BPEL is a workflow-like definition language that allows to describe sophisticated business processes; WS-Coordination and WS-Transaction complement it to provide mechanisms for defining specific standard protocols to be used by transaction processing systems, workflow systems, or other applications that wish to coordinate multiple services. Together, these specifications address connectivity issues that arise when Web Services run on several platforms across organizations.

1.2 Transactions in Web Services

A common business scenario involves multiple parties and different organizations over a time frame. Negotiations, commitments, shipments and errors happen. A business transaction between a manufacturer and its suppliers ends successfully only when parts are delivered to their final destination, and this could be days or weeks after the initial placement of the order.

A transaction completes successfully (*commits*) or it fails (*aborts*) undoing (*roll-backing*) all its past actions. Web services transactions [17] are long-running transactions. As such, they pose several problems. It is not feasible to turn an entire long-running transaction into an ACID transaction, since maintaining isolation for a long time poses performance issues [31]. Roll-backing is also an issue. Undoing many actions after a long time from the start of a transaction entails trashing what could be a vast amount of work.

Since in our scenario a traditional roll-back is not feasible, Web Services orchestration environments provide a *compensation* mechanism which can be executed when the effects of a transaction must be cancelled. What a compensation policy does depends on the application. For example, a customer orders a book from an on-line retailer. The following day, that customer gets a copy of the book elsewhere, then requests the store to withdraw the order. As a compensation, the store can cancel the order, or charge a fee. In any case, in the end the application has reached a state that it considers equivalent to what it was before the transaction started.

The notions of orchestration and compensation require a formal definition. In this paper, we address orchestration with particular attention to web transactions. We introduce $\texttt{web}\pi_\infty$, a subcalculus of $\texttt{web}\pi$ [15] that does not model time, as a simple extension of the π-calculus. As a case study, we discuss and formalize an e-commerce transactional scenario building on a previous one, which we presented in an earlier work [12] using a different algebra, the Event Calculus, which we introduced in [18]. The Event Calculus needed some improvement to make it more readable and easier to use for modelling real-world scenarios. This paper is a step in that direction.

1.3 Related Work

In this paper we mainly refer to BPEL, the most likely candidate to become a standard among workflow-based composition languages. Other languages have

been introduced, among them WS-CDL [14], which claims to be in some relation with the fusion calculus [22].

Other papers discuss formal semantics of compensable activities in this context. [13] is mainly inspired by XLANG; the calculus in [9] is inspired by BP-Beans [10]; the πt-calculus [8] focuses on BizTalk; [6] deals with short-lived transactions in BizTalk; [7] also presents the formal semantics for a hierarchy of transactional calculi with increasing expressiveness.

Some authors believe that time should be introduced both at the model level and at the protocols and implementation levels [15, 3, 2, 4]. XLANG, for instance, provides a notion of *timed transaction* as a special case of long running activity. BPEL uses timers to achieve a similar behavior. This is a very appropriate feature when programming business services which cannot wait forever for the other parties reply.

1.4 Outline

This work is organized as follows. In Section 2 we explain our formal approach to orchestration: extending the π-calculus to include transactions. In Section 3 we discuss this extension with its syntax and semantics, while in Section 4 we discuss an e-commerce transactional scenario to show the strength of the language. Section 5 draws a conclusion.

2 A Formal Approach to Web Services Orchestration

Business process orchestration has to meet several requirements, including providing a way to manage exceptions and transactional integrity [23]. Orchestration languages for Web Services should have the following interesting operations: sequence, parallel, conditional, send to/receive from other Web Services on typed WSDL ports, invocation of Web Services, error handling.

BPEL covers all these aspects. Its current specification, however, is rather involved. A major issue is error handling. BPEL provides three different mechanisms for coping with abnormal situations: *fault handling*, *compensation handling* and *event handling*. [1] Documentation shows ambiguities, in particular when interactions between these mechanisms are required. Therefore it is difficult to use the language, and we want to address this issue.

Our goal is to define a clear model with the smallest set of operators which implement the operations discussed above, and simple to use for application designers. We build on the π-calculus [21, 20, 24], a well known process algebra. It is simple and appropriate for orchestration purposes. It includes: a parallel operator allowing explicit concurrency; a restriction operator allowing compositionality and explicit resource creation; a recursion or a process definition operator allowing Turing completeness; a sequence operator allowing causal relationship

[1] The BPEL event handling mechanism was not designed for error handling only. However, here we use it for this purpose.

between activities; an inaction operator which is just a ground term for inductive definition on sequencing; message passing and in particular name passing operators allowing communication and link mobility.

There is an open debate on the use of π-calculus versus Petri nets in the context of Web Services composition [27]. The main reason here for using the π-calculus for formalization is that the so called *Web Services composition languages*, like XLANG, BPEL and WS-CDL claim to be based on it, and they should therefore allow rigorous mathematical treatment. However, no interesting relation with process algebras has really been proved for any of them, nor an effective tool for analysis and reasoning, either theoretical or software based, has been released. Therefore, we see a gap that needs to be filled, and we want to address the problem of composing services starting directly from the π-calculus.

By itself the π-calculus does not support any transactional mechanism. Programming complex business processes with failure handling in term of message passing only is not reasonable; also, the Web Services environment requires that several operations have transactional properties and be treated as a single logical unit of work when performed within a single business transaction. Below we consider a simple extension of the π-calculus that covers transactions.

3 The Orchestration Calculus $\texttt{web}\boldsymbol{\pi}_{\boldsymbol{\infty}}$

The syntax of $\texttt{web}\pi_\infty$ *processes* relies on countable sets of *names*, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\widetilde{u}$.

$$
\begin{array}{lll}
P ::= & & \\
 & \mathbf{0} & \text{(nil)} \\
 & \mid \overline{x}\,\langle \widetilde{u} \rangle & \text{(output)} \\
 & \mid x(\widetilde{u}).P & \text{(input)} \\
 & \mid (x)P & \text{(restriction)} \\
 & \mid P \mid P & \text{(parallel composition)} \\
 & \mid A(\tilde{u}) & \text{(process invocation)} \\
 & \mid \langle\!|P \ ; \ P|\!\rangle_x & \text{(transaction)}
\end{array}
$$

We are assuming a set of process constants, ranged over by A, in order to support process definition. A defining equation for a process identifier A is of the form

$$A(\tilde{u}) \stackrel{def}{=} P$$

where each occurrence of A in P has to be guarded, i.e. it is underneath an input prefix. It holds $fn(P) \subseteq \{\tilde{u}\}$ and $\tilde{u}$ is composed by pairwise distinct names.

A process can be the inert process $\mathbf{0}$, an output $\overline{x}\,\langle \widetilde{u} \rangle$ sent on a name x that carries a tuple of names $\widetilde{u}$, an input $x(\widetilde{u}).P$ that consumes a message $\overline{x}\,\langle \widetilde{w} \rangle$ and behaves like $P\{^{\widetilde{w}}/_{\widetilde{u}}\}$, a restriction $(x)P$ that behaves as P except that inputs and messages on x are prohibited, a parallel composition of processes, a process invocation $A(\tilde{u})$ or a transaction $\langle\!|P \ ; \ R|\!\rangle_x$ that behaves as the *body* P until a transaction abort message $\overline{x}\,\langle\rangle$ is received, then it behaves as the *compensation* Q.

Names x in outputs and inputs are called *subjects* of outputs and inputs respectively. It is worth noticing that the syntax of $\texttt{web}\pi_\infty$ processes simply extends the asynchronous π-calculus with the transaction process.

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

$\texttt{fn}(\langle\!|P\ ;\ R|\!\rangle_x) = \texttt{fn}(P) \cup \texttt{fn}(R) \cup \{x\}$ and α-equivalence equates $(x)(\langle\!|P\ ;\ Q|\!\rangle_x)$ with $(z)(\langle\!|P\{^z/_x\}\ ;\ Q\{^z/_x\}|\!\rangle_z)$;

In the following we let $\tau.P$ be the process $(z)(\overline{z}\,\langle\rangle \mid z().P)$ where $z \notin \texttt{fn}(P)$. $\texttt{web}\pi_\infty$ processes considered in this paper are always *well-formed* according to the following:

Definition 1 (Well-formedness). *Received names cannot be used as subjects of inputs. Formally, in $x(\widetilde{u}).P$ free subjects of inputs in P do not belong to names $\widetilde{u}$.*

This property avoids a situation where different services receive information on the same channel, which is a nonsense in the service oriented paradigm.

3.1 Semantics of the Language

We give the semantics for the language in two steps, following the approach of Milner [19], separating the laws which govern the static relations between processes from the laws which 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 2. *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:*

1. *the scope laws:*

$$(u)\mathbf{0} \equiv \mathbf{0}, \qquad (u)(v)P \equiv (v)(u)P,$$
$$P \mid (u)Q \equiv (u)(P \mid Q)\ , \quad \text{if } u \notin \texttt{fn}(P)$$
$$\langle\!|(z)P\ ;\ Q|\!\rangle_x \equiv (z)\langle\!|P\ ;\ Q|\!\rangle_x\ , \quad \text{if } z \notin \{x\} \cup \texttt{fn}(Q)$$

2. *the invocation law:*

$$A(\tilde{v}) \equiv P\{\tilde{v}/\tilde{u}\} \qquad \text{if } A(\tilde{u}) \stackrel{def}{=} P$$

3. *the transaction laws:*

$$\langle\!|\, \mathbf{0} \;;\; Q \,|\!\rangle_x \equiv \mathbf{0}$$
$$\langle\!|\, \langle\!|\, P \;;\; Q \,|\!\rangle_y \mid R \;;\; R' \,|\!\rangle_x \equiv \langle\!|\, P \;;\; Q \,|\!\rangle_y \mid \langle\!|\, R \;;\; R' \,|\!\rangle_x$$

4. *the floating law:*

$$\langle\!|\, \overline{z}\,\langle\widetilde{u}\rangle \mid P \;;\; Q \,|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|\, P \;;\; Q \,|\!\rangle_x$$

The scope and invocation laws are standard. Let us discuss transaction and floating laws, which are unusual. The law $\langle\!|\, \mathbf{0} \;;\; Q \,|\!\rangle_x \equiv \mathbf{0}$ defines committed transactions, namely transactions with $\mathbf{0}$ as body. These transactions, 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 transactions outside parent transactions, thus flattening the nesting of transactions. Notwithstanding this flattening, parent transactions may still affect children transactions by means of transaction names. The law $\langle\!|\, \overline{z}\,\langle\widetilde{u}\rangle \mid P \;;\; R \,|\!\rangle_x \equiv \overline{z}\,\langle\widetilde{u}\rangle \mid \langle\!|\, P \;;\; R \,|\!\rangle_x$ floats messages outside transactions; it models that 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 transaction boundaries, until it reaches the corresponding input. In case an outer transaction fails, recovery actions for this message may be detailed inside the compensation processes. The dynamic behavior of processes is defined by the reduction relation.

Definition 3. *The* reduction relation $\rightarrow$ *is the least relation satisfying the following axioms and closed with respect to* $\equiv$, $(x)\_\ ,\ \_\,|\,\_$ *and* $\langle\!|\ \_\ ;\ Q \,|\!\rangle_x$*:*

$$\overline{x}\,\langle\widetilde{v}\rangle \mid x(\widetilde{u}).P \;\stackrel{(\textsc{com})}{\rightarrow}\; P\{\widetilde{v}/\widetilde{u}\}$$
$$\overline{x} \mid \langle\!|\, \textstyle\prod_{i\in I} x_i(\widetilde{u_i}).P_i \;;\; Q \,|\!\rangle_x \;\stackrel{(\textsc{fail})}{\rightarrow}\; Q \qquad (I \neq \emptyset)$$

Rule (com) is standard in process calculi and models input-output interaction. Rule (fail) models transaction failures: when a transaction abort (a message on a transaction name) is emitted, the corresponding transaction is terminated by garbage collecting the threads (the input processes) in its body and activating the compensation. On the contrary, aborts are not possible if the transaction is already terminated, namely every thread in the body has completed its job.

4 A Case Study

In this section, we discuss an implementation in $\texttt{web}\pi_\infty$ of a classical e-business scenario: a customer attempts to buy a set of items from some providers, using a coordination service exposed by a web portal. Actors involved in this e-business scenario are a *customer*, a *web portal* and a set of *item providers*.

4.1 Participants

The roles who take part in the purchase scenario are the following:

1. a **customer** sends a request to a shopping portal, and waits for a response. The customer can express some constraints: for example, "I want to buy either all items or no one at all". The web portal takes care of implementing policies like this one;
2. a **web portal** tries to fulfill customers' requests and their constraints about the purchase policy. It acts as a coordinator;
3. an **item provider** accepts two kinds of requests from the web portal: a simple browsing of the price-list (read-only), and a purchase request of an item.

The web portal, on behalf of a customer, tries to buy an item from a provider. This could be a failure or success. In case of failure, the web portal is informed, and the item provider forgets everything about the transaction. In case of success, if the request can be fulfilled, the item provider declares that the sale is complete, and it begins the execution of an internal process which simulates the delivery of the item. Meanwhile, the customer can change her mind and tell the item provider, which will *compensate* the relative transaction, i.e. take some actions to establish a safe state. An example of compensation may be charging a fee. This mechanism will be explained more in detail below within the $\texttt{web}\pi_\infty$ specification.

4.2 Constraints

When sending a purchase request, a customer can also specify the behavior that the complete transaction must follow. For example, a customer wants to buy formal attire: a suit, a pair of shoes, a shirt and a tie. A reasonable constraint to impose is that either the shirt and the tie should come together, or none of them, while the suit and the shoes are optional. In our specification, we describe a simplified policy called *all or nothing*. This means that the purchase transaction will be successful only if all sub-transactions will commit, otherwise the purchase will fail. To implement this constraint, the web portal uses the compensation service that the item providers provide.

Buy requests are emitted simultaneously to each item provider, and the web portal gets their outcomes. If each sub-transaction is successful, the web portal informs the customer that its request has been satisfied, otherwise, it compensates any committed sub-transaction.

In our implementation we simplify this scenario. Instead of asking the customer for constraints over an order, we apply a built-in policy. This is fair to pose, because constraints are contained in the coordinator process, and this does not affect the behavior of item providers. It is also very easy to specify different purchase policies, because they are clearly separated from the mechanisms which control them. Further, we also assume that a customer wants to buy two items only from two different sellers.

4.3 Formal Description

We now present a formal description of all participants and how they can be composed in an e-business scenario.

World

$$\begin{aligned}\mathsf{WORLD}() := (a_c)(a_p)(c_1)(p_1)(c_2)(p_2)(\\ &\mathsf{CUSTOMER}(a_c, a_p)\\ &|\,\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)\\ &|\,\mathsf{IP}_1(c_1, p_1)\\ &|\,\mathsf{IP}_2(c_2, p_2))\end{aligned}$$

The process $\mathsf{WORLD}()$ composes the various participants to the scenario; first of all, it creates some global channels, used by the processes to interact together: the channels a_c and a_p are the web portal interfaces exposed to the customers. So, they are passed as arguments both to the $\mathsf{CUSTOMER}(a_c, a_p)$ and to the $\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)$ processes. The first one is used to require a price list, while the second one to emit a purchase order.

The other global channels are the set of pairs c_i and p_i, which are respectively the query and the purchase interface of the i^{th} item provider. Those names are passed as arguments to the $\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2)$ and $\mathsf{IP_i}(c_i, p_i)$ processes.

We do not model message loss, because we suppose that reliable protocols are used, which would take care of any transmission error, and we ignore the issue of site crashes. We also assume the world as a closed system, in the sense that $\mathtt{fn}(()WORLD()) = \emptyset$. Because of the dynamic nature of the scenario, this could be regarded as a rather strong assumption. All these aspects could be taken into account in a future evolution of the specification.

Customer

$$\begin{aligned}\mathsf{CUSTOMER}(a_c, a_p) := (\widetilde{q_1})(\widetilde{q_2})(a_r)(a_s)(a_f)(\\ &\overline{a_c}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_r\rangle \,|\, a_r(\widetilde{l_1}, \widetilde{l_2}).\overline{a_p}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_s, a_f\rangle \,|\\ &a_s().\mathsf{S}() \,|\, a_f().\mathsf{F}())\end{aligned}$$

The customer process first browses a price list. When it receives an answer, it emits a purchase request, and waits for the outcome. To do this it creates these names: $\widetilde{q_1}$ and $\widetilde{q_2}$, which contain the two item preferences, the channel a_r, which is the restricted reply channel used by the Web Portal to inform the customer about the price list consultation, and the two channels a_s (success) and a_f (failure), which signal respectively the outcome of the purchase transaction. Then the customer process sends the message $\overline{a_c}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_r\rangle$ to the web portal consultation interface. This message carries the items description and the reply channel. This first phase ends with the receipt of the reply message $a_r(\widetilde{l_1}, \widetilde{l_2})$, which carries two names, $\widetilde{l_1}$ and $\widetilde{l_2}$, encoding the features of the requested items, like their availability, the selling price and many others. Basing on this information, the customer process elaborates its orders — which are encoded in $\widetilde{q_1}$ and $\widetilde{q_2}$ — and sends a purchase order $\overline{a_p}\,\langle \widetilde{q_1}, \widetilde{q_2}, a_s, a_f\rangle$ containing the item specifications and

the outcome channels, a_s and a_f. When the customer process receives one of this message, the purchase transaction has completed and it goes on with the appropriate task identified by $\mathsf{S}()$ or $\mathsf{F}()$. Moreover, it is guaranteed that either all the items have been bought, or the appropriate compensations have been emitted.

Web Portal

$$
\begin{aligned}
&\mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2) := a_c(\widetilde{q}_1, \widetilde{q}_2, a_r).(\\
&\qquad \mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r) \mid\\
&\qquad \mathsf{WEB\_PORTAL}(a_c, a_p, c_1, p_1, c_2, p_2))\\
&\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r) := \mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r) \mid\\
&\qquad \mathsf{PURCHASE}(a_p, p_1, p_2)\\
&\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r) := (r_1)(r_2)(\overline{c_1}\,\langle \widetilde{q}_1, r_1\rangle \mid \overline{c_2}\,\langle \widetilde{q}_2, r_2\rangle \mid\\
&\qquad r_1(\widetilde{q}_1, \widetilde{l}_1).r_2(\widetilde{q}_2, \widetilde{l}_2).\overline{a_r}\,\langle \widetilde{l}_1, \widetilde{l}_2\rangle)\\
&\mathsf{PURCHASE}(a_p, p_1, p_2) := a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f).(r_s^1)(r_f^1)(r_s^2)(r_f^2)(\\
&\qquad \overline{p_1}\,\langle \widetilde{q}_1, r_s^1, r_f^1\rangle \mid \overline{p_2}\,\langle \widetilde{q}_2, r_s^2, r_f^2\rangle \mid\\
&\qquad \mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f))
\end{aligned}
$$

The web portal process exposes a service which can be used by a customer to query some distributed price lists, and subsequently to purchase the items. When it receives a request $a_c(\widetilde{q}_1, \widetilde{q}_2, a_r)$, it executes a managing process — $\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ — and it creates a duplicate, to wait for further requests.

The $\mathsf{ENGINE}(a_p, c_1, p_1, c_2, p_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ process executes two sub-processes $\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$ and $\mathsf{PURCHASE}(a_p, p_1, p_2)$. The first of these subtasks, $\mathsf{QUERY}(c_1, c_2, \widetilde{q}_1, \widetilde{q}_2, a_r)$, receives the consulting channels c_1 and c_2, the customer preferences $\widetilde{q}_1$ and $\widetilde{q}_2$ and the reply channel a_r. It emits in parallel the various price list consultations with the messages $\overline{c_1}\,\langle \widetilde{q}_1, r_1\rangle$ and $\overline{c_2}\,\langle \widetilde{q}_2, r_2\rangle$, which contain the customer preferences and the private channels r_1 and r_2 on which it will wait for a reply. Those replies contain the outcomes of the queries executed on the item provider's databases — encoded with names $\widetilde{l}_1$ and $\widetilde{l}_2$. When the web portal receives them, it forwards them to the customer application with the message $\overline{a_r}\,\langle \widetilde{l}_1, \widetilde{l}_2\rangle$, and it waits for a purchase order on the channel $a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f)$.

The process $\mathsf{PURCHASE}(a_p, p_1, p_2)$ is called with the channel a_p, on which it will wait for the customer's order, and the item providers' channels p_1 and p_2. First, it receives the customer's request $a_p(\widetilde{q}_1, \widetilde{q}_2, a_s, a_f)$, which contains the item specifications and the pair of success/failure channels. At this point, it creates a pair of success/failure reply channels r_s and r_f for each item provider, and emits the purchase requests $\overline{p_1}\,\langle \widetilde{q}_1, r_s^1, r_f^1\rangle$ and $\overline{p_2}\,\langle \widetilde{q}_2, r_s^2, r_f^2\rangle$. When the requests have been emitted, the process $\mathsf{PURCHASE}(a_p, p_1, p_2)$ executes the process $\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f)$, which will manage the purchase transactions' outcomes.

Waiting Process. The process $\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f)$ waits for the outcome of the item provider 1 in this way:

$$
\begin{aligned}
\mathsf{WAIT}(r_s^1, r_f^1, r_s^2, r_f^2, a_s, a_f) &:= r_s^1(\widetilde{q}_1, \widetilde{l}_1, t_1).\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f) \\
&\quad \mid r_f^1(\widetilde{q}_1, \widetilde{l}_1).\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f) \\
\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f) &:= r_s^2(\widetilde{q}_2, \widetilde{l}_2, t_2).\mathsf{POLICY}_{\mathsf{S},\mathsf{S}}(t_1, t_2, a_s, a_f) \\
&\quad \mid r_f^2(\widetilde{q}_2, \widetilde{l}_2).\mathsf{POLICY}_{\mathsf{S},\mathsf{F}}(t_1, a_s, a_f) \\
\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f) &:= r_s^2(\widetilde{q}_2, \widetilde{l}_2, t_2).\mathsf{POLICY}_{\mathsf{F},\mathsf{S}}(t_2, a_s, a_f) \\
&\quad \mid r_f^2(\widetilde{q}_2, \widetilde{l}_2).\mathsf{POLICY}_{\mathsf{F},\mathsf{F}}(a_s, a_f)
\end{aligned}
$$

If the item provider 1 is able to fulfill the order, it emits a message on the input channel $r_s^1(\widetilde{q}_1, \widetilde{l}_1, t_1)$. When the web portal receives this message, the process $\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f)$ can start. This process manages all the cases in which the item provider 1 is successful. On the other hand, if the item provider 1 is not able to fulfill the order, the web portal receives a failure message on the input channel $r_f^1(\widetilde{q}_1, \widetilde{l}_1)$, and the process $\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f)$ is executed. This process manages all the cases in which the item provider 1 fails.

The behavior of $\mathsf{WAIT}_{\mathsf{S},\star}(t_1, r_s^2, r_f^2, a_s, a_f)$ and $\mathsf{WAIT}_{\mathsf{F},\star}(r_s^2, r_f^2, a_s, a_f)$ is quite clear: each one waits for the outcome of the item provider 2. When the web portal receives the message, it will be alternatively in one of four possible states, as shown in figure 1.

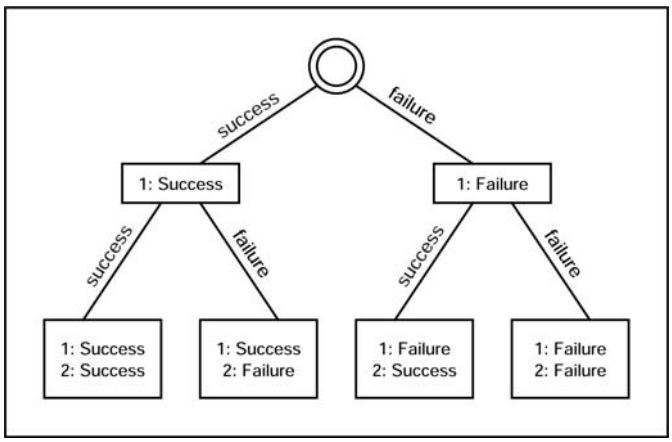

Fig. 1. Tree of Possible Executions

Policy Process. When all outcome messages have been collected, the web portal is able to take the appropriate actions: this is done by the following processes:

$$
\begin{aligned}
\mathsf{POLICY}_{\mathsf{S},\mathsf{S}}(t_1, t_2, a_s, a_f) &:= \overline{a_s}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{S},\mathsf{F}}(t_1, a_s, a_f) &:= \overline{a_f}\,\langle\rangle \mid \overline{t_1}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{F},\mathsf{S}}(t_2, a_s, a_f) &:= \overline{a_f}\,\langle\rangle \mid \overline{t_2}\,\langle\rangle \\
\mathsf{POLICY}_{\mathsf{F},\mathsf{F}}(a_s, a_f) &:= \overline{a_f}\,\langle\rangle
\end{aligned}
$$

The first process manages the case in which both of the item providers are successful; in this case, the customer is informed that its purchase order can

be fulfilled. This process receives the compensation handlers t_1 and t_2 also if it does not use them. This is because, in general, the web portal could implement a policy different from *all or nothing*.

The process $\mathsf{POLICY}_{\mathsf{S},\mathsf{F}}(t_1, a_s, a_f)$ manages the case where the item provider 1 is successful, and the item provider 2 is faulty: to fulfill the constraints imposed by the customer, the transaction is cancelled with the emission of the compensation request $\overline{t_1}\,\langle\rangle$. This way, the web portal implements the *all or nothing* behavior required by the customer. The case where the item provider 1 is faulty while the item provider 2 is successful is simply the dual case. The case where both the item providers are faulty is managed simply by emitting a message on the reply channel a_f, and no compensation is required.

It would be easy to generalize the algorithm to an *at least one* policy. In such a scenario, the web portal would send a success message in all the first three cases, while in the fourth one, it would send a failure message. No compensations would be required.

Item Provider

$$\mathsf{IP_i}(c_i, p_i) := (db_c)(db_p)(\mathsf{CP_i}(c_i, db_c) \mid \mathsf{PP_i}(p_i, db_p) \mid \mathsf{DBP_i}(db_c, db_p))$$

The generic i^{th} item provider receives two names as arguments, c_i and p_i. These names are global, i.e. they have been created by the $\mathsf{WORLD}()$ process. The former represent the item provider interface for the consulting service, while the latter is used to receive a buying order. When the item provider process begins its execution, it creates a pair of channels, which are used to interact with a database process. The channel db_c is used to invoke a price list consultation service exposed by the database; the channel db_p is used to emit a purchase order to the same database. After the creation of these channels, the item provider creates three sub-processes, $\mathsf{CP_i}(c_i, db_c)$, $\mathsf{PP_i}(p_i, db_p)$ and $\mathsf{DBP_i}(db_c, db_p)$. The first two processes manage the consultation and the purchase orders emitted by the customer, while the third one represents a database process.

Consulting Process

$$\begin{aligned}\mathsf{CP_i}(c_i, db_c) := {} & c_i(\widetilde{q_i}, r_i).((odbc)(\overline{db_c}\,\langle \widetilde{q_i}, odbc\rangle \mid odbc(\widetilde{q_i}, \widetilde{l_i}).\overline{r_i}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle) \\ & \mid \mathsf{CP_i}(c_i, db_c))\end{aligned}$$

$\mathsf{CP_i}(c_i, db_c)$ is a server process which receives price list read requests. It receives two names, c_i and db_c. The first name is the input channel it will listen to for a request, while the second one is the access point for the database querying service. The process $\mathsf{CP_i}(c_i, db_c)$ behaves as follows: when it receives a price check request $c_i(\widetilde{q_i}, r_i)$, containing the customer preferences $\widetilde{q_i}$ and a reply channel r_i, it duplicates itself and begins the price list reading operations. It creates a fresh name, *odbc*, and sends it to the database consulting service with the message $\overline{db_c}\,\langle \widetilde{q_i}, odbc\rangle$, which contains also the customer preferences $\widetilde{q_i}$. Then it waits for an outcome $(odbc(\widetilde{q_i}, \widetilde{l_i}))$ and forwards it to the web portal, using the reply channel $\overline{r_i}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle$.

Purchase Process

$$\begin{aligned}\mathsf{PP_i}(p_i, db_p) := p_i(\widetilde{q_i}, r_s, r_f).((odbc_s)(odbc_f)(s)(f)(t_i)(\overline{db_p}\,\langle \widetilde{q_i}, odbc_s, odbc_f, s\rangle \\ \mid \{\!|odbc_s(\widetilde{q_i}, \widetilde{l_i}, t).(\overline{f}\,\langle\rangle \mid \overline{r_s}\,\langle \widetilde{q_i}, \widetilde{l_i}, t_i\rangle \mid t_i().\overline{t}\,\langle\rangle)\ ;\ \mathbf{0}|\!\}_s \\ \mid \{\!|odbc_f(\widetilde{q_i}, \widetilde{l_i}).(\overline{s}\,\langle\rangle \mid \overline{r_f}\,\langle \widetilde{q_i}, \widetilde{l_i}\rangle)\ ;\ \mathbf{0}|\!\}_f) \mid \mathsf{PP_i}(p_i, db_p))\end{aligned}$$

The second sub-process created by the item provider $\mathsf{PP_i}(p_i, db_p)$ manages the purchase orders emitted by the web portal on behalf of the customer. When this process runs, it receives two names, p_i and db_p. The first name is the access point for the purchase service exposed by the item provider. The second name represents a private channel shared between the purchase manager and the database process that is used to invoke the purchase service exposed by the database.

The process $\mathsf{PP_i}(p_i, db_p)$ waits for a purchase request on the global channel p_i. The request contains the customer's preferences $\widetilde{q_i}$ and a pair of success/failure reply channels, r_s and r_f. When the process receives this message, it makes a copy of itself and waits for further requests, and begins the purchase managing operations. First it creates two fresh names, $odbc_s$ and $odbc_f$, which are a pair of success/failure reply channels. Then it creates two transactions, s and f, which manage the cases of success and failure of the purchase process. Those names are restricted, together with the name t_i, which will be used by the web portal to compensate a successful purchase transaction. The purchase process emits a request message $\overline{db_p}\,\langle \widetilde{q_i}, odbc_s, odbc_f, s\rangle$, which contains the customer preferences $\widetilde{q_i}$, a pair of success/failure reply channels $odbc_s$ and $odbc_f$ and the name of successful transaction manager, s. Its usefulness is shown below.

After the emission of the purchase request, the process activates the success and the failure transactions. Those transactions share a very similar behavior. Each one listens to the appropriate channel for the database outcome. This means that the transaction s waits for a success message on the $odbc_s$ channel, while the transaction f waits on the $odbc_f$ channel. In both cases, the outcome message brings the customer preferences $\widetilde{q_i}$ and the query result $\widetilde{l_i}$. Moreover, in case of success, the message contains also the name of the database transaction which manages the delivery of the requested item. This name can be used to compensate this activity, as we show below.

When one of the two specular transaction receives the purchase outcome, it triggers the other one. As the two compensation processes are the **0** process, this mechanism acts like an explicit garbage collector.[2] After receiving of the outcome, the appropriate transaction forwards it to the web portal. In case of a success, moreover, the reply message contains also a transaction name that can be used to activate the database delivery compensation. Instead of the original name received by the database process, t, a placeholder, t_i, is sent. This forbids a direct access to an internal process — the database — by an external process. In case of success, indeed, the item provider acts as a wrapper for the database

[2] This feature is not really necessary, because the other transaction remains deadlocked on a restricted name, but is useful to show how it is possible to implement a garbage collector with the compensation mechanism provided by the transactions.

compensation mechanism. When the item provider receives a compensation request, it emits the correct signal. The execution of this wrapper process lasts until the delivery operations end. When this happens the clearing signal s is emitted by the database process.

Database Process

$$
\begin{array}{lll}
\mathsf{DBP_i}(db_c, db_p) & := & \mathsf{DBP_i^c}(db_c) \mid \mathsf{DBP_i^p}(db_p) \\
\mathsf{DBP_i^c}(db_c) & := & db_c(\widetilde{q}_i, odbc).((\widetilde{l}_i)(\overline{odbc}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle) \mid \mathsf{DBP_i^c}(db_c)) \\
\mathsf{DBP_i^p}(db_p) & := & db_p(\widetilde{q}_i, odbc_s, odbc_f, s).(\\
 & & (\widetilde{l}_i)(t)(\overline{odbc_s}\,\langle \widetilde{q}_i, \widetilde{l}_i, t \rangle \mid (\langle\!| dlv() \ ; \ cmp() |\!\rangle_t.\overline{s}\,\langle\rangle) \\
 & & \oplus \overline{odbc_f}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle) \mid \\
 & & \mathsf{DBP_i^p}(db_p))
\end{array}
$$

The third sub-process created by the item provider is $\mathsf{DBP_i}(db_c, db_p)$. This process simulates the behavior of a DBMS. In particular, it exposes two kinds of services: the price list consultation and the purchase order. It receives a pair of private channels db_c and db_p and shares them with the item provider. The former is the access point on which it will wait for a price list consultation, while the latter is used to listen for purchase orders.

Two distinct sub-processes manage the two activities mentioned above. The process $\mathsf{DBP_i^c}(db_c)$ manages the price list consultation. When it receives a request message, it creates a duplicate. The request message carries the customer's preferences $\widetilde{q}_i$ and a reply channel $odbc$. Now, the database simply creates a new name, $\widetilde{l}_i$, which represents the outcome of the query executed on the DBMS, and sends it back to the item provider. This operation simulates a database query, and can never fail; if a query produces no results, its outcome is correctly encoded on the fresh name $\widetilde{l}_i$.

The process $\mathsf{DBP_i^p}(db_p)$ deals with purchase orders, delivery of goods and any compensation requested by the web portal. At first, the process receives a purchase order from the item provider. This request contains the item preferences $\widetilde{q}_i$, a pair of success/failure reply channels $odbc_s$ and $odbc_f$ and a transaction name s. When it receives the request, the process makes a copy of itself, creates a new name $\widetilde{l}_i$, which represents the query outcome, and decides if the customer's request can be fulfilled or it must be rejected. To do so, it uses a constructor called *internal choice*, which is represented with the symbol $\oplus$. This means that only one process is chosen, while the other is simply discharged. This behavior is easily encodable in terms of parallel composition, message passing and restriction only. We introduce this notation just for brevity.

If the database purchase process is not able to fulfill the order, it simply emits a message $\overline{odbc_f}\,\langle \widetilde{q}_i, \widetilde{l}_i \rangle$ on the failure reply channel $odbc_f$, and forgets everything about the transaction. The message contains the customer's preferences $\widetilde{q}_i$ and the outcome of the query, represented by $\widetilde{l}_i$. In case of item availability, the behavior of the database process is more complex. On the successful channel $odbc_s$, it emits a reply message, which contains the customer preferences $\widetilde{q}_i$, the outcome of the query $\widetilde{l}_i$ and the compensation handler t. In parallel with the reply message emission, the database process begins to execute the delivery

operations. From this moment on, the web portal can emit the compensation request while the delivery action is being performed.

5 Conclusion

In this paper we introduced $\texttt{web}\pi_\infty$, a simple extension of the π-calculus with untimed long running transactions. We discussed the notion of orchestration without considering time constraints. This way we focused on information flow, message passing, concurrency and resource mobility, keeping the model small and simple. We motivated the underlying theory we rely on, the π-calculus, in terms of expressiveness and suitability to composition and orchestration purposes. To show the strength of the language we also proposed a formalization of an e-commerce transactional scenario.

This work contributes a simple, concise yet powerful and expressive language, with a solid semantics that allows formal reasoning. The language shows a clear relation with the π-calculus, and the actual encoding of it with the π-calculus is a feasible task, while it would be quite harder to to get such an encoding for XLANG and other Web Services composition languages.

A possible extension of this work could be generalizing the transaction policy and proving constraints satisfaction. Other future developments building on the results achieved in this paper include software tools for static analysis of programs using composition and orchestration. A useful result that could stem from this work could be streamlined definitions of syntax and semantics of web services composition languages, to get a simpler way to model involved transaction behaviors. On a more theoretical side, another research direction could be extending the calculus with a notion of time while keeping it simple. The overall goal we have is to allow for improvement of quality and applicability of real orchestration languages.

Acknowledgments. The authors would like to acknowledge Cosimo Laneve, Enrico Tosi and Andrea Carpineti for their comments and contributions to the paper.

References

1. T. Andrews, F. Curbera et al. Web Service Business Process Execution Language, Working Draft, Version 2.0, 1 December 2004.
2. M. Berger. Basic Theory of Reduction Congruence for Two Timed Asynchronous π-calculi. In *CONCUR'04: Proceedings of the 15th International Conference on Concurrency Theory, LNCS* 3170, pages 115-130, Springer-Verlag, 2004.
3. M. Berger. Towards Abstractions for Distributed Systems. PhD Thesis, Imperial College, London, 2002.
4. M. Berger, K. Honda, The Two-Phase Commit Protocol in an Extended π-Calculus. In *EXPRESS '00: Proceedings of the 7th International Workshop on Expressiveness in Concurrency, ENTCS* 39.1, Elsevier, 2000.

5. Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp], Microsoft Corporation.
6. R. Bruni, C. Laneve, U. Montanari. Orchestrating Transactions in Join Calculus. In *CONCUR'02: Proceedings of the 13th International Conference on Concurrency Theory*, *LNCS* 2421, pages 321-337, Springer-Verlag, 2003.
7. R. Bruni, H. Melgratti, U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. To appear in POPL2005.
8. L. Bocchi, C. Laneve, G. Zavattaro. A Calculus for Long-running Transactions. In *FMOODS'03: Proceedings of th 6th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems*, *LNCS* 2884, pages 124-138, Springer-Verlag, 2003.
9. M. Butler, 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*, *LNCS* 2949, pages 87-104. Springer-Verlag, 2004.
10. M. Chessel, D. Vines, C. Griffin, V. Green, K. Warr. Business Process Beans: System Design and Architecture Document. Technical report.IBM UK Laboratories. January 2001.
11. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL 1.1). [www.w3.org/TR/wdsl], W3C, Note 15, 2001.
12. C. Guidi, R. Lucchi, M.Mazzara. A Formal Framework for Web Services Coordination. 3rd International Workshop on Foundations of Coordination Languages and Software Architectures, London 2004.
13. T. Hoare. Long-Running Transactions. Powerpoint presentation [research.microsoft.com/]
14. N. Kavantzas, G. Olsson, J. Mischkinsky, M. Chapman. Web Services Choreography Description Languages. [otn.oracle.com/tech/webservices/ htdocs/spec/cdl_v1.0.pdf]
15. C. Laneve, G. Zavattaro. Foundations of Web Transactions. To appear in FOSSACS 2005.
16. F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.
17. M. Little. Web Services Transactions: Past, Present and Future. [www.idealliance.org/papers/dx_xml03/ html/abstract/05-02-02.html]
18. M.Mazzara, R.Lucchi. A Framework for Generic Error Handling in Business Processes. First International Workshop on Web Services and Formal Methods (WSFM), Pisa 2004.
19. R. Milner. Function as Processes. *Mathematical Structures in Computer Science*, 2(2):119-141, 1992.
20. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
21. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.
22. J. Parrow, B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *LICS'98: Proceedings of the 13th Symposium on Logic in Computer Science*, IEEE Computer Society Press.
23. C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, October 2003 (Vol.36, No 10), pages 46-52.
24. D. Sangiorgi, D. Walker. *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

25. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.
26. Universal Description, Discovery and Integration for Web Services (UDDI) V3 Specification. [http://uddi.org/pubs/uddiv3.htm]
27. W.M.P. van der Aalst. Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". [mitwww.tm.tue.nl/staff/wvdaalst/publications/pi-hype.pdf]
28. Workflow Management Coalition - http://www.wfmc.org/
29. WS-Coordination Specification [www-106.ibm.com/developerworks/library/ws-coor/]
30. WS-Transaction Specification [www-106.ibm.com/developerworks/webservices/library/ws-transpec/]
31. B. Weikum, G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.