

From Theory to Practice in Transactional Composition of Web Services

Roberto Bruni¹, Gianluigi Ferrari¹, Hernán Melgratti¹, Ugo Montanari¹,
Daniele Strollo², and Emilio Tuosto¹

¹ Dipartimento di Informatica,
Università degli Studi di Pisa, Italy
email: {bruni,giangi,melgratt,ugo,etuosto}@di.unipi.it

² Istituto Alti Studi IMT Lucca, Italy
email: daniele.strollo@imtlucca.it

Abstract We address the problem of composing Web Services in long-running transactional business processes, where compensations must be dealt with appropriately. Long-running transactions differ from typical data-bases transactions since data-bases relies on perfect rollback mechanisms.

The framework presented in this paper is a Java API called *Java Transactional Web Services* (JTWS), which provides suitable primitives for wrapping and invoking Web Services as activities in long-running transactions. JTWS adheres to a process calculi formalisation of long-running transactions, called Naïve Sagas, which fixes unambiguously the implemented compensation policy. In particular, the primitives provided by JTWS are in one-to-one correspondence with the primitives of Sagas, and they are abstract enough to hide the complex details of their realization, thus favouring usability. Moreover, JTWS orchestrates business processes in a distributed way.

1 Introduction

One of the emerging issues when aggregating *Web Services* (WSs, for short) is constituted by the so-called *long-running transactions* (LRTs, for short), i.e. the possibility of requiring a set of WSs interactions to be executed atomically. Note that the problem is not just to coordinate the updates of a distributed repository (e.g., a database), since components are independent and any of them is responsible for maintaining the consistency on local data. A distinguishing feature of LRTs is the fact that atomicity relies on *compensations*, namely, ad-hoc activities that are responsible for undoing the effects of partial executions when the overall orchestration cannot be completed. In fact, most of the languages proposed in recent years for orchestrating WSs (e.g., WSCL [28], WSCI [27], BPML [5], WSFL [23], XLANG [29], BPEL4WS [3]) include primitives for handling LRTs. Noteworthy, all those proposals formalise the orchestration syntax but not the semantics, whose informal description can make the intended behaviour of constructs ambiguous and can lead to different implementations of the same language. (As an example, see the large list of open issues for BPEL4WS [4].) Moreover, those proposals mix together many different concepts and programming constructs. Hence, a clear

semantics for them is difficult to be established because one has to consider the mutual interactions of such different constructs.

In this paper, we first take advantage of a formal framework for isolating and studying LRTs and then we use an experimental framework for implementing and exercising the theoretical choices in a WSs scenario. We are aimed at building a framework for coordinating transactional compositions of WSs over a solid formal basis. The main goal of our work is two-fold. Firstly, we provide application designers with a formally specified language for defining transactional aggregations at a high level of abstraction, i.e. in terms of the involved WSs and the control flow among them regardless of low level details, e.g. distribution, asynchrony, etc. Secondly, after selecting a coordination infrastructure, we map high level transactional primitives into concrete orchestration patterns. One of the main advantages of our approach lies in the reciprocal benefits that theory and practice can gain in this case. For instance, several alternative semantics can be given when composing parallel transactional flows [6]. The possibility of giving prototype implementations of those semantics and apply them to realistic scenarios (like WS applications) can help in evaluating and modifying the theoretical models. Moreover, applications deployed with the aid of formal methods are more robust and can be more easily analysed.

The high level language we choose is Naïve Sagas [8], a process calculus for compensable transactions, while the orchestration infrastructure we propose is *Java Transactional Web Services* (JTWS) [25], an execution platform that supports the transactional capabilities of Naïve Sagas. Indeed, JTWS embeds the transactional policies of Naïve Sagas into a framework for programming WSs.

From the existing calculi for LRTs [13,12,14,16,8,2,21,7,15,18], we have chosen Naïve Sagas because we find it particularly convenient to expose the orchestration mechanism behind LTRs by abstracting away from the computation performed by the involved WSs. In fact, activities in a saga are described at a high level of abstraction, mostly in the style of CCS, where the elementary actions are not interpreted. Transactional flows basically are processes built by composing with the standard parallel and sequential composition plus the *compensation pair* construct. Given two actions A and B , the compensation pair $A \div B$ corresponds to a process that uses B as compensation of A . Intuitively, $A \div B$ yields two flows of execution: the *forward flow* and the *backward flow*. During the forward flow, $A \div B$ starts its execution by running A and then, when A finishes: (i) B is “installed” as compensation of A , and (ii) the control is forwardly propagated to the other stages of the transactions. In case of failure in the rest of the transaction, the backward flow starts so that the effects of A ’s execution must be rolled back. This is achieved by executing the installed compensation B and afterward by propagating the rollback to the previously executed activities. Note that B is not installed if A is not executed.

The execution platform JTWS is a Java implementation of the APIs defined in [25,20]. JTWS is based on a signal passing style of programming. Conceptually, JTWS is divided in two levels: *Java Signal Core Layer* (JSCL) and *Java Transactional Layer* (JTL). The former provides a set of primitives for defining and handling the flow of signals among components. The latter, uses the primitives of JSCL to define the behaviour of transactional constructs according to Naïve Sagas. Basically, WSs are wrapped into JTWS

components that exchange a fixed set of suitable signals. Similarly, JTWS fixes a precise flow of signals for composed services. It is worth remarking that the JSCL layer provides a general framework for implementing different transactional policies. Indeed, one can easily change the behaviours of transactions by replacing the JTL layer. Therefore, we can prototype and experiment with different semantics for transactional flows without changing the code of the application.

Related works. Several process calculi have been proposed to deal with different flavours of transactions. Notably, models for ACID (i.e., usual database transactions) transactions in Linda [17] have been proposed in [1,11,10,19,9]. We prefer Sagas to the ACID models because ACID transactions are regarded as not suitable for computations that may elapse for a long period of time. Similarly, our work differs in scope from [18], which is aimed at extending an object oriented programming language with primitives for handling ACID transactions.

Another mainstream in transactional process calculi takes as starting point well-known name passing calculi, like π and Join, and adds to them transactional features like compensable nested contexts [2], timed transactions [21], interacting compensable transactions [7]. We prefer Sagas to those approaches because Sagas naturally abstracts away from low level computations and communication patterns, while it highlights the composition structure of transactional processes. Similarly, we prefer Sagas to approaches like [15], where the coordination mechanism relies on the operations performs over a centralised log.

Sagas is much more in the spirit of StAc [12,13] and cCSP [14]. (We refer to [6] for a detailed comparison of both approaches.) We prefer Sagas to those proposals because it is more compact and can be easily extended with additional features.

As far as the execution platform is concerned, our approach is different from the existing implementations of orchestration languages that we are aware of (like Biztalk [22], Oracle BPEL Process Manager [24] and WebSphere [26]) because JTWS does not rely on an engine that rules the execution of a composed process. Instead, it provides WSs with the coordination infrastructure needed to wire services into a larger process but keeping the coordination logic distributed.

2 Background: sagas calculus

In this section we introduce the formal basis for the implementation of the JTL package. In particular, we exploit a process algebra for compensable flow composition that is essentially the algebra of Naïve Sagas in [8], but whose semantics is here presented in the simpler style of compensating CSP (cCSP) [14].

2.1 Syntax

Sagas are built over a set of atomic activities $\Sigma \cup \{0, THROW\}$, where 0 (the nil activity) and *THROW* (the interrupting activity) are two distinguished elements. Atomic activities are ranged over by *A, B, ...*

The set of processes is defined as follows:

COMPOSITION OF STANDARD TRACES

$$\begin{array}{l}
\mathbf{Sequential} \begin{cases} p\langle\checkmark\rangle; s = ps \\ p\langle\omega\rangle; s = p\langle\omega\rangle \text{ when } \omega \neq \checkmark \end{cases} \\
\mathbf{Parallel} \quad p\langle\omega\rangle || q\langle\omega'\rangle = \{r\langle\omega\&\omega'\rangle \mid r \in \text{int}(p, q)\}, \quad \text{where} \quad \frac{\omega \quad \begin{array}{c} | \\ \hline \end{array} \begin{array}{c} ! \\ ! \\ ! \\ ? \\ ? \\ ? \\ \checkmark \end{array}}{\omega' \quad \begin{array}{c} | \\ \hline \end{array} \begin{array}{c} ! \\ ? \\ \checkmark \\ ? \\ \checkmark \\ \checkmark \end{array}} \\
\quad \quad \quad \frac{\omega \quad \omega' \quad \begin{array}{c} | \\ \hline \end{array} \begin{array}{c} ! \\ ! \\ ! \\ ? \\ ? \\ \checkmark \end{array}}{\omega\&\omega' \quad \begin{array}{c} | \\ \hline \end{array} \begin{array}{c} ! \\ ! \\ ! \\ ? \\ ? \\ \checkmark \end{array}} \\
\text{and} \quad \begin{cases} \text{int}(p, \langle \rangle) = \{p\} \\ \text{int}(\langle \rangle, q) = \{q\} \\ \text{int}(\langle x \rangle p, \langle y \rangle q) = \{\langle x \rangle r \mid r \in \text{int}(p, \langle y \rangle q)\} \cup \{\langle y \rangle r \mid r \in \text{int}(\langle x \rangle p, q)\} \end{cases}
\end{array}$$

TRACES OF NAÏVE SAGAS

$$\begin{array}{l}
\Gamma \vdash \quad 0 = \{\langle\checkmark\rangle\} \\
\Gamma \vdash \text{THROW} = \{\langle!\rangle\} \\
\Gamma \vdash \quad A = \{\langle A, \checkmark \rangle\} \quad \text{when } A \in \Sigma \wedge \Gamma(A) = \checkmark \\
\Gamma \vdash \quad A = \{\langle!\rangle\} \quad \text{when } A \in \Sigma \wedge \Gamma(A) = ! \\
\Gamma \vdash \quad S; T = \{s; t \mid \Gamma \vdash s \in S \wedge \Gamma \vdash t \in T\} \\
\Gamma \vdash \quad S|T = \{s' \mid s' \in (s||t) \wedge \Gamma \vdash s \in S \wedge \Gamma \vdash t \in T\} \\
\Gamma \vdash \quad [P] = \{p\langle\checkmark\rangle \mid \Gamma \vdash (p\langle\checkmark\rangle, s) \in P\} \cup \{ps \mid \Gamma \vdash (p\langle!\rangle, s) \in P\}
\end{array}$$

Figure 1. Trace semantics of Naïve Sagas

$$(\text{NAÏVE SAGAS}) \quad S, T ::= A \mid [P] \mid S; T \mid S|T$$

$$(\text{COMPENSABLE PROCESSES}) \quad P, Q ::= A \div B \mid P; Q \mid P|Q$$

A Naïve Sagas is either a basic activity A , a transaction block enclosing a compensable process $[P]$, the sequential composition $S; T$ of sagas, or the parallel composition $S|T$ of sagas. A basic compensable process is a compensation pair $A \div B$ where A is a basic activity and B is its compensation. Compensable processes can be composed either in sequence $P; Q$ or in parallel $P|Q$.

Without loss of generality, we assume that any activity in Σ appears at most once in any saga (resp. process), i.e. that different instances of the same action are named differently.

2.2 Semantics

The semantics is defined in terms of admissible execution traces. A trace for a saga is a string $s\langle\omega\rangle$, where $s \in \Sigma^*$ is said the *observable flow* and $\omega \in \Omega$ is the *final event*, with $\Omega = \{\checkmark, !, ?\}$ (\checkmark stands for success, $!$ for fail, and $?$ for yielding to a concurrent interrupt and it is assumed that $\Sigma \cap \Omega = \emptyset$). Hereafter, we let p, q, r range over Σ^* and s, t range over the set of traces $\Sigma^*\Omega$. The sequential composition $s; t$ concatenates the observable flows of s and t only when s terminates with success, otherwise it is s . The composition of two concurrent traces $p\langle\omega\rangle || q\langle\omega'\rangle$ corresponds to the set $\text{int}(p, q)$ of all possible interleaving of the observable flows followed by the final event $\omega\&\omega'$, where

COMPOSITION OF COMPENSABLE TRACES

$$\begin{array}{l}
\mathbf{Comp. pair} \begin{cases} p\langle\checkmark\rangle \div s = (p\langle\checkmark\rangle, s) \\ p\langle\omega\rangle \div s = (p\langle\omega\rangle, \langle\checkmark\rangle) \text{ when } \omega \neq \checkmark \end{cases} \\
\mathbf{Sequential} \begin{cases} (p\langle\checkmark\rangle, s); (t, t') = (pt, t'; s) \\ (p\langle\omega\rangle, s); (t, t') = (p\langle\omega\rangle, s) \text{ when } \omega \neq \checkmark \end{cases} \\
\mathbf{Parallel} \begin{cases} (p\langle\checkmark\rangle, s) || (q\langle\checkmark\rangle, t) = \{(r\langle\checkmark\rangle, s') \mid r \in \text{int}(p, q) \wedge s' \in (s || t)\} \\ \quad \cup \{(r\langle?\rangle, \langle\omega\rangle) \mid r\langle\omega\rangle \in (ps || qt)\} \\ (p\langle\omega\rangle, s) || (q\langle\omega'\rangle, t) = \{(r\langle\omega \& \omega'\rangle, \langle\omega''\rangle) \mid r\langle\omega''\rangle \in (ps || qt)\} \\ \quad \text{when } \omega \& \omega' \in \{!, ?\} \end{cases}
\end{array}$$

TRACES OF COMPENSABLE PROCESSES

$$\begin{array}{l}
\Gamma \vdash A \div B = \{s \div t \mid \Gamma \vdash s \in A \wedge \Gamma \vdash t \in B\} \\
\Gamma \vdash P; Q = \{s; t \mid \Gamma \vdash s \in P \wedge \Gamma \vdash t \in Q\} \\
\Gamma \vdash P|Q = \{s' \mid \Gamma \vdash s' \in (s || t) \wedge \Gamma \vdash s \in P \wedge \Gamma \vdash t \in Q\}
\end{array}$$

Figure 2. Trace semantics of compensable processes

the associative and commutative operator $\&$ defines the final event corresponding to the parallel composition of traces. The set of traces is evaluated according to a scenario $\Gamma : \Sigma \rightarrow \{\checkmark, !\}$ decreeing the success or failure of each basic activity. We write $\Gamma \vdash S = S'$ if S and S' represent the same set of traces under the scenario Γ . We write $\Gamma \vdash s \in S$ if the set of traces associated to S under the scenario Γ includes s .

Figure 1 summarises the trace semantics of Naïve Sagas. The definition for the traces of sagas is straightforward. The most interesting definition is for a transaction block $[P]$. Note that any trace of a compensable process P is a pair $(p\langle\omega\rangle, s)$, where $p\langle\omega\rangle$ is the forward trace and s is the corresponding compensation trace. Then, the definition for $[P]$ selects all successful traces of P (i.e., $p\langle\checkmark\rangle$), and the traces corresponding to the failed forward flows followed by their compensations, i.e., ps . A compensated trace ps ending with \checkmark corresponds to an aborted execution that has been compensated successfully. Instead, if the compensated trace finishes with $!$, then the execution of some compensation failed. We refer to the latter case as to an execution that raises an exception. Moreover, note that a trace that finishes with \checkmark has not enough information to distinguishing whether it corresponds to the successful execution of the forward flow (i.e., a commit) or to a successfully compensated flow (i.e., an abort with a complete compensation). Note that all pairs whose forward traces end with $?$ are just discarded.

Figure 2 gives the semantics of compensable processes. The traces of a compensation pair are just given by the pairs of traces for the forward and backward flows, but the compensation is installed only if the forward activity ends with success. When composing compensable traces in series, the forward trace corresponds to the sequential composition of the original forward traces, while the compensation trace starts by the second compensation followed by the first one, so that compensations are executed in the reverse order w.r.t. the associated forward activities. The parallel composition is defined as suitable interleaving of the forward and the backward flows. The parallel

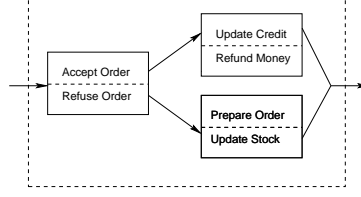


Figure 3. A parallel saga for handling orders

composition of two successful traces contains all the interleaving of the forward flows compensated with the interleaving of the original compensations, and a set of yielding traces. Yielding traces stand for the behaviours of processes $P|Q$ in case they are composed in parallel with a process that fails, for instance $P|Q|THROW$. Note that this is the only case where yielding behaviours are generated in the semantics (in particular, neither backward traces, nor standard traces can ever contain the final event $?$). Finally, the parallel composition when at least one trace ends with $?$ or $!$ is defined as the interleaving of the original compensated flows.

The trace semantics can be used to prove interesting laws which hold under every scenario. We write $S \equiv T$ if for all Γ we have $\Gamma \vdash S = T$. For example, it can be readily proved that sequential and parallel compositions of sagas (resp. compensable processes) are associative and commutative under any scenario Γ . Other easy equivalences are:

$$\begin{array}{ll}
 0;P \equiv P;0 \equiv P & THROW;P \equiv THROW \\
 0;S \equiv S;0 \equiv S|0 \equiv S & THROW;S \equiv THROW \\
 THROW \div A \equiv THROW \div 0 & [A \div A'|B \div B'|THROW] \equiv (A;A')|(B;B')
 \end{array}$$

Although a more significant example is presented in Section 5, we describe here a small example illustrating the features of Naïve Sagas.

Example 1 (Handling Purchase Orders). Consider the simple business process for handling purchase orders depicted in Figure 3. The first activity *Accept Order* handles a request from a client and it is compensated by *Refuse Order*, which will contact the client to notify her/him that the order was cancelled. After that, both the balance of the client's account is updated and the order is prepared. The step *Update Credit* charges the amount of the order to the balance of the client. This activity could fail, for instance when the client has not enough credit to proceed, which will activate the compensation installed so far (i.e., *Refuse Order*). Instead, if it succeeds, then the compensation *Refund Money* is also installed. *Refund Money* is responsible for updating the balance with the amount detracted previously. Finally, *Prepare Order* handles the packaging of the order and updates the stock. Its compensation *Update Stock* will increment the stock with the proper values. Using the obvious acronyms in place of activities, the saga for handling purchase orders can be written as

$$\text{HPO-saga} \stackrel{\text{def}}{=} [A.O. \div R.O.; (U.C. \div R.M.|P.O. \div U.S.)]$$

In a scenario Γ in which all activities are successful, the set of traces will be

$$\Gamma \vdash \text{HPO-saga} = \{ \langle A.O., U.C., P.O., \checkmark \rangle, \langle A.O., P.O., U.C., \checkmark \rangle \}$$

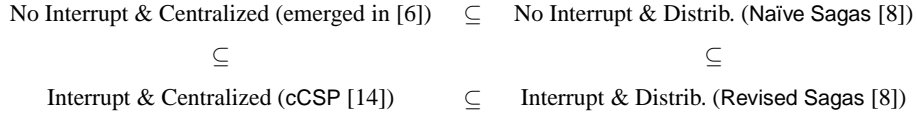


Figure 4. Compensation Policies for Parallel Flows

Instead, in a scenario Γ' like Γ but where the client has not enough credit to proceed, the activity Update Credit fails and thus

$$\Gamma' \vdash \text{HPO-saga} = \{\langle \text{A.O.}, \text{P.O.}, \text{U.S.}, \text{R.O.}, \checkmark \rangle\}$$

As a last scenario, consider Γ'' like Γ' but where upon failure of Update Credit the compensation Update Stock of the activity Prepare Order fails because the goods cannot be unpackaged without damage. Then, the saga will raise an exception:

$$\Gamma'' \vdash \text{HPO-saga} = \{\langle \text{A.O.}, \text{P.O.}, ! \rangle\}$$

2.3 Discussion

The calculus we have presented is obtained by mixing the ingredients coming from two different proposals [8,14]. For example, the use of scenarios comes from [8], while the interleaving trace semantics is more in the style of [14]. For the sake of presentation, we focus here just on parallel sagas by leaving out several other features considered in [8,14], like exception handling, choices, and nesting.

The integration the two approaches is sustained by the detailed comparison carried out in [6], where sagas [8] and cCSP [14] are reconciled. In particular, it is shown that for the sequential composition both approaches coincide in the way in which compensations are installed and activated, while different compensation policies are used for parallel composition. In fact, [8] proposes already two different semantics for parallel compensations, called *naïve* and *revised*. Nevertheless, none of them coincides with the one in [14]. The key difference lies in the activation of sibling compensations in parallel branches of a transaction when one of the branches compensates. In fact there are several policies for notifying the abort to sibling processes. Roughly, such policies can be characterised in terms of two orthogonal strategies: (i) whether the forward flow can be interrupted to activate the compensation procedure as soon as possible or not; and (ii) whether the compensation procedure is activated in a centralised or in a distributed way. The combination of these strategies gives the four different policies shown in Figure 4. The main result in [6] is to relate the different semantics arising in the four cases, which justifies the inclusion relations depicted in Figure 4. Suitable counterexamples for proving that Naïve Sagas $\not\subseteq$ cCSP and cCSP $\not\subseteq$ Naïve Sagas are given in [6].

The four strategies mentioned above correspond to alternative implementations for the compensation mechanism. The policy adopted when presenting the semantics in Figures 1 and 2 is *no interruption and distributed compensation*, a distributed procedure

for compensating parallel branches that may allow the execution of activities of the backward flow even when parts of siblings forward flow are still in execution. As an example, the aforementioned law $[A \div A' \mid B \div B' \mid THROW] \equiv A;A' \mid B;B'$ illustrates this policy. Note that the forward flow is executed completely (i.e., A and B) but parallel branches are independently compensated for, e.g. A' can be executed even before B completes.

We conclude this section by remarking that our calculus is tailored to Naïve Sagas, and hence some syntactic assumptions and the semantics in Figures 1 and 2 are slightly different w.r.t. the presentation in [6], where a uniform style of description for the four policies has been preferred.

3 Java Transactional Web Services

In this section we describe JTWS, a Java-implementation of Naïve Sagas based on the APIs introduce in [25,20]. The programming pattern adopted in JTWS is based on signal passing. Basically, WSs become JTWS components which interact by exchanging suitable signals. It is possible to divide JTWS in two conceptual levels, JSCL and JTL: the former provides the signal handling primitives that the latter uses to define the transactional ones. Hereafter, JTWS components are called *gates*.

Signal core layer. The signal layer JSCL abstracts the primitive mechanisms for defining and exchanging signals among gates.

A signal represents an event that occurs on a given gate, for instance, a service may notify a successful execution by emitting a suitable signal. Like in the event-notification pattern, “handlers” are associated signals. Handlers must subscribe in order to be notified and are not necessarily unique, i.e., several handlers are possibly associated to a signal. Unlike the event-notification pattern, JSCL gates behave as handlers and emitters for different signals at the same time.

The class SIGNAL defines the JSCL signals which carry some *internal information* (e.g., sender/receiver identifiers, synchronous/asynchronous, timestamp...) and *session data*. The session data can be accessed by invoking the methods for getting/setting the session attributes (e.g., GETPARAM, SETPARAMVALUE, ID). (For a detailed presentation of the whole session and internal data APIs, the reader is referred to [25]. Details therein are not necessary to understand the rest of the paper.)

Conceptually, a generic JTWS gate, graphically represented in Figure 5(a), controls its internal resources and communicates with other gates by means of I/O ports where signals of a specified type can be received/sent. Ports are an idealisation, indeed in JTWS they are not effectively implemented as objects, but are characterised by the signal types. Therefore, signals having different types corresponds to signals sent on different ports. All the types of the input (or output) ports of a gate are pairwise distinct.

Ports are connected through *links* which carry signals from emitters to handlers. The links in JSCL are typed, unidirectional and unicast (namely, links connect a single emitter to a single handler). More complex scenarios (e.g., multi-casting, bi-directionality, etc.) can be obtained by opportunely connecting links. For instance, multi-casting is achieved by connecting the same emitter to several handlers (with as many links as the handlers). As for the ports, links are not explicitly implemented in JTWS; links

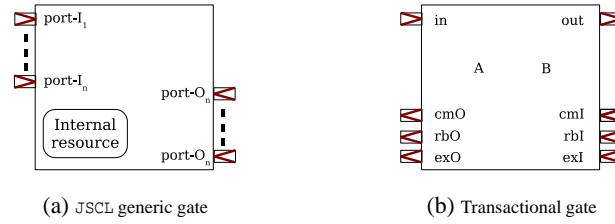


Figure 5. Gates

are effectively represented by writing the information on the handlers in the internal data of the signal. The emitter e can create a link toward the handler h by executing $e.CREATELINK(t, h)$ which also specifies the type t of the link. Afterward, e can emit signals toward h with the `EMITSIGNAL` method provided by the API. If many links exist between e and h , the type of the emitted signal is used to determine which is the actual link to be used.

Transactional layer. JTL represents a specialisation of JSCL focused on describing the transactional aspects related to the forward and backward flows across gates. This level fixes the set of the signals that gates may exchange and their semantics. According to Naïve Sagas, JTL gates must handle four flows of execution, which are represented by the signals `FORWARD`, `COMMIT`, `ROLLBACK` and `EXCEPTION`. All of them are exchanged on specific ports (see Section 4 for details).

The signal `FORWARD` encodes the forward flow of sagas and implements the normal execution. The remaining signals encode the backward flows. More precisely, `COMMIT` corresponds to the flow of the correct termination of a saga, while `ROLLBACK` and `EXCEPTION` detect the failures of the saga. Indeed, `ROLLBACK` encodes the flow of execution that start when a normal execution fails and `EXCEPTION` encodes the flow starting when a rollback flow fails.

A *transactional gate* (shortly *TG*) can be defined by specialising the class `TRANSACTIONALCOMPONENT` and implementing the methods `ONFORWARD`, `ONCOMMIT`, `ONROLLBACK` and `ONEXCEPTION` which handle the corresponding signals. The mechanism to emit a signal is inherited from JSCL. The classes `TRANSACTIONALSEQUENCE` and `TRANSACTIONALPARALLEL` provide the methods for creating transactional gates by sequential and parallel composition of *TGs*. Their public interfaces remain the same as the one of *TG* so that they can be inductively composed, as shown in Sections 4.1 and 4.2. Moreover, they are equipped with the method `addInternalComponent` that allows to add a new gate to an existing sequential or parallel gate.

4 Sagas in JTWS

The main ingredients of the translation from Naïve Sagas to JTWS are: (1) the signals representing the states of the transactions, (2) the atomic tasks representing the atomic actions of Naïve Sagas and (3) the transactional generic gates corresponding to sagas.

The signal `EXCEPTION` implements the final event `'!`' while `ROLLBACK` and `COMMIT` are the `JTWS` counterpart of `'✓'`, the final event decorating the traces that successfully terminate either their forward or backward flows. Furthermore, `ROLLBACK` is used as the (internal) signal for starting the compensation of a saga. Moreover, we also use a `COMMIT` signal representing the normal termination of transactional gates. Hereafter, it is assumed that signal emissions do not fail.

At the `JSQL` level, the method `onHandleSignal` is overridden so that the appropriate method is invoked when a signal is received. For instance, when a `ROLLBACK` is emitted, the `onHandleSignal` of all the registered handler is called so that the method associated to it (i.e., `onRollback`) is invoked.

4.1 Gates for compensation pairs and sequential composition

Transactional gates are objects of the class `TG` and are the building blocks of sagas, which are obtained by gluing together transactional gates such that the interface and the behaviour on them are preserved. Basically, a transactional gate is a generalisation of the elementary step $A \div B$, graphically represented in Figure 5(b). The atomic actions A and B are implemented as objects `A` and `B`, respectively, in a class implementing the interface `AtomicTask`:

```
public interface AtomicTask {
    public abstract Object execute (Signal signal) throws AtomicActionException;
}
```

where we assume that an `AtomicTask` object starts its execution when its `execute` method is invoked on a signal and throws an exception if a failure occurs.

The function `comp(A,B)` records A and B into a transactional gate as illustrated in Figure 5(b). Intuitively, A starts its execution when a `FORWARD` signal is received on the `in` port. Commenting on Figure 6, when the `onForward` method is executed, the gate tries to run A ; whenever the execution of A normally terminates, the `FORWARD` signal is propagated. On the contrary, if A throws an exception, a `ROLLBACK` is emitted. The other methods act similarly. If A normally terminates its execution, `comp(A,B)` forwards on the `out` port the signal for invoking the next gates in the saga. When a `COMMIT` signal is received on the `cmI` port, the gate `comp(A,B)` forwards it on the `cmO` port.

If an exception occurs during the execution of A , then `comp(A,B)` emits a `ROLLBACK` on port `rbO` so that the gates waiting for the result of the saga can compensate. The alternative anomalous execution is constituted when receiving an exception signal on the port `exI`. This signal is emitted when an exception occurs during the execution of a compensation. For instance, if B catches an exception, it emits on its `exO` port the `EXCEPTION` signal. Hence, the gate that receives it will backwardly propagate `EXCEPTION` to the other gates of the saga.

Transactional gates can be sequentially composed in an easy manner as illustrated in Figure 7(a). An arrow from a port to another represents a link for a specific type of signals from the emitter to the handlers. Hence, all the target gates of the arrow have been registered as handlers for that signal. For instance, in Figure 7(a), the signals emitted from Q on its `cmO` are handled by P , which receives them on the `cmI` port. As explained in Section 3, the ports are associated with the proper method that should be

```

public class Step extend GenericTransactionalGate {
    private AtomicTask A = null;
    private AtomicTask B = null;

    public comp(AtomicTask A, AtomicTask B) {this.A = A; this.B = B }

    public int onForward(Signal signal) {
        try {
            if (A != null) then A.execute(signal);
            signal.setType(SignalType.FORWARD);
            emitSignal(signal);
        } catch (AtomicActionException e) {
            signal.setType(SignalType.ROLLBACK);
            emitSignal(signal);
        }
    }

    public int onRollback(Signal signal) {
        try {
            if (B != null) then B.execute(signal);
            signal.setType(SignalType.ROLLBACK);
            emitSignal(signal);
        } catch (AtomicActionException e) {
            signal.setType(SignalType.EXCEPTION);
            emitSignal(signal);
        }
    }

    public int onCommit(Signal signal) {
        signal.setType(SignalType.COMMIT);
        emitSignal(signal);
    }

    public int onException(Signal signal) {
        try {
            signal.setType(SignalType.EXCEPTION);
            emitSignal(signal);
        } catch (Object e) {
            emitSignal(signal);
        }
    }
}

```

Figure 6. The class GenericTransactionalGate

executed when a signal is received on them; for instance, signals received on `rbI` are associated with the `onRollback` method that activates the compensation of the gate.

Given two transactional gates P and Q , $\text{seq}(P, Q)$ yields the transactional gate having as `in`, `cmO`, `rbO` and `exO` ports those of P , while the ports `out`, `cmI`, `rbI` and `exI` are those of Q . The behaviour of $\text{seq}(P, Q)$ is as follows: any FORWARD signal received from the `in` port of P is sent on the `out` port of P if it normally terminates, otherwise a ROLLBACK is sent on the `cmO` port of P . Gate Q after receiving the FORWARD (i.e., P has executed correctly) executes as any other gate. Notice that, in case Q fails its normal execution, the ROLLBACK is handled by P , which starts its compensation and either backwardly propagates the ROLLBACK (if the compensation normally terminates) on the port `rbO` of P or emits an EXCEPTION signal on the port `exO` of P (if any failure occur during the compensation of P).

Signals received from Q on its `cmI` and `exI` ports are simply propagated (on the corresponding ports) to P , which backwardly propagates them to the rest of the saga.

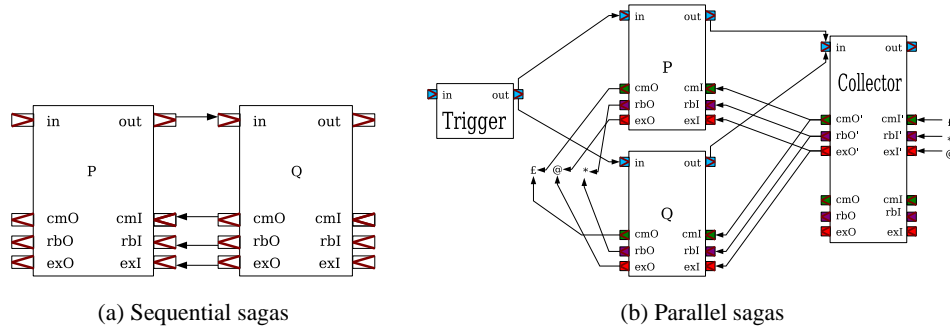


Figure 7. Composition of sagas

4.2 Gates for parallel composition and sagas

The behaviour of a parallel saga is the most complex among the coordination constructs of Naïve Sagas. The function `par` yields the transactional gate obtained by composing in parallel a (finite) number of transactional gates. For simplicity, we illustrate `par` in the case of the parallel composition of two gates P and Q , as illustrated in Figure 7(b) (the case where more than two gates are composed in parallel is analogous and equivalent to compose $P \mid Q$ with R).

Function `par` uses two auxiliary gates: the `Trigger` and the `Collector`. These gates are transparent to the users. Note that the `Trigger` and the `Collector` are not transactional gates themselves. On the invocation signal, the `Trigger` simply triggers all the gates of the parallel saga. The `Collector` manages the result of the parallel saga and interfaces the intermediate results of P and Q with the transactional gates outside the saga. As shown in Figure 7(b), `Collector` has the ports for receiving/sending signals to P and Q and those for external gates `cmI`, `cmO`, `rbI`, `rbO`, etc.

In Figure 7(b) we represent the links between `Trigger` and `Collector` of a parallel saga made of two transactional gates P and Q .

The parallel saga is activated when `Trigger` sends the `FORWARD` received from its `in` port to its `out` port. According to our assumption, P and Q start their executions. At this point several cases are possible depending on the results from P , Q and the signals from the gates external to the parallel saga. In order to forward the invocation signal, the `Collector` waits for the `invoke` signals from P and Q . When those signals are received, `Collector` waits for the rest of the saga to communicate the result.

1. If a `COMMIT` is received on port `cmI`, then it is forwarded on `cmO'` to P and Q which forward it on their `cmO` ports to `Collector`. Once all the commit signals are collected, a `COMMIT` is emitted on the port `cmO` of `Collector`.
2. If a `ROLLBACK` is received on port `rbI`, then it is forwarded on `rbO'` to P and Q which activate their compensations. At this point, either P and Q signal a `ROLLBACK` on their `rbO` ports or one of them emits a `exception` on `exO`. In the former case, analogously to the previous case, the rollback is propagated on the `rbO` port of

Collector. If one of P or Q (or both of them) emits an exception, then Collector propagates an exception signal on exO instead of a rollback one on cmO .

3. If an EXCEPTION is received on port exI , then it is forwarded on rbO' to P and Q (which activate their compensations). At this point, the signals from P and Q are ignored and Collector emits an exception signal on exO .

Either P or Q might fail their normal execution and emits a ROLLBACK signal. Consider that P emits a ROLLBACK signal. In this case, the simplest scenario is when also Q emits a ROLLBACK: Collector will simply emit a ROLLBACK of its rbO port. If Q sends an FORWARD signal, as soon as Collector receives it replies with a ROLLBACK for Q . Afterward, if Q sends the second ROLLBACK, Collector proceeds as in the previous case; on the contrary, Q might reply with an EXCEPTION signal. In this case, Collector signals an exception on its exO port.

The function $\text{closure}(P)$ allows one to consider the saga as a method invocation. This is obtained by simply connecting the out port of the gate P with its cmI port. When the saga is invoked, a signal is sent on the in port of P . The control is returned when P emits a signal either on port cmO or on rbO . An exception is raised if P emits a signal on port exO .

We close this section by summarising the mapping from Naïve Sagas to JTWS as follows:

$$\begin{array}{ll} \llbracket A \div B \rrbracket = \text{comp}(A, B) & \llbracket P; Q \rrbracket = \text{seq}(P, Q), \\ \llbracket P|Q \rrbracket = \text{par}(P, Q) & \llbracket [P] \rrbracket = \text{closure}(P) \end{array}$$

where A (resp. B) is the `AtomicTask` object encoding A (resp. B) and P (resp. Q) is the transactional gate implementing the compensable process P (resp. Q).

5 A Case Study

In this section, we exemplify the use of JTWS to accomplish the task of providing transactional behaviour to WSs composition. Our case study scenario will focus on the development of an application combining two overlay networks, namely the Internet and a telecommunication network. The application provides a SMS Taxi Booking facility. The basic idea of the application is that registered customers can book a taxi by sending a SMS text message to the Taxi call-centre. The customer gets a SMS reply back from the taxi company confirming the booking along with the estimated arrival time, place, fare and vehicle details. Moreover, the full amount of the fare at the end of the journey will be payed on-line by exploiting the registered information about customer credit card. This application has been designed, deployed, and executed within a framework that integrates web services, a rich set of telecommunication services (including call/session control, messaging features, presence and location features) and web services for telecommunications (Parlay X Web services) [25]. Our aim is to show the adequacy of JTWS (and the underlying process calculus Naïve Sagas) for designing LRTs. Indeed, our case study offers a test-bed for the programming features of JTWS.

Figure 8 pictorially illustrates the overall structure of the application. The SMS Taxi Booking service is structured into three stages. The first stage treats the taxi booking activities. The second stage manages the communications for the confirmation of

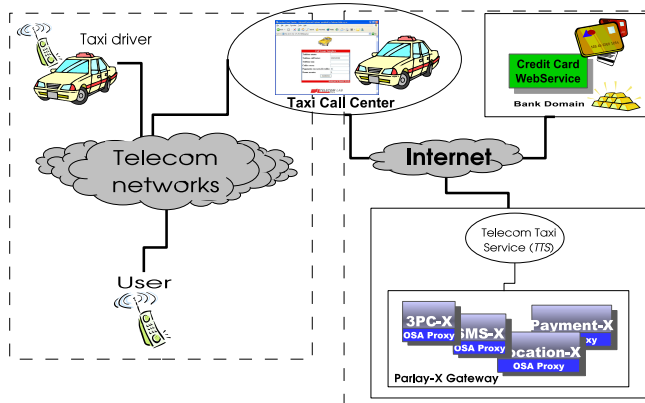


Figure 8. SMS Taxi Booking Service

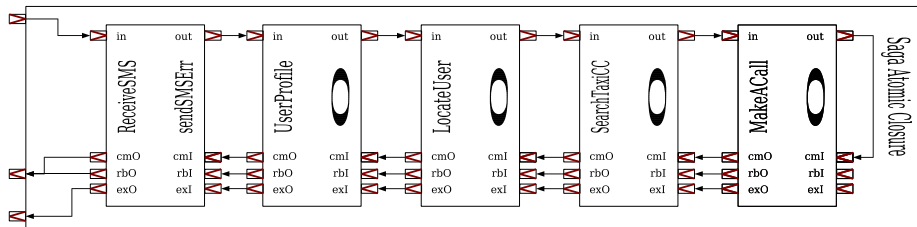


Figure 9. Stage 1: JTWS Description

the booking. Finally, the last stage handles the taxi payment service. We focus on the implementation of the first and third stages, which involve non-trivial transactional facets. Figure 9 sketches the JTWS representation of the first stage. The saga implementing the first stage is the sequential composition of several services. The `ReceiveSMS` service is the access gateway of the application and is activated upon receipt of the SMS message. Its main activity consists in generating the activation signal for all the other services, which check whether the customer is authorised to access the service (`UserProfile`), determine the location of the user¹ (`LocateUser`), select the Taxi Company (`SearchTaxiCC`) and finally set up a call between the taxi company and the customer (`MakeACall`). Note that the `ReceiveSMS` compensation (`sendsMSSErr`) is indeed the only compensation of the whole sequence: it will be executed in case of the failure of the booking (an appropriate error message will be sent to the user). The compensations of the other services are all empty, indeed none of them modifies the local state and their failures just activate the emission of the `ROLLBACK` signal.

The saga in Figure 10 describes the implementation of the third stage of the application. Intuitively, after having retrieved the reservation data, the services for the

¹ User localisation is obtained by exploiting the features of the telecommunication services.

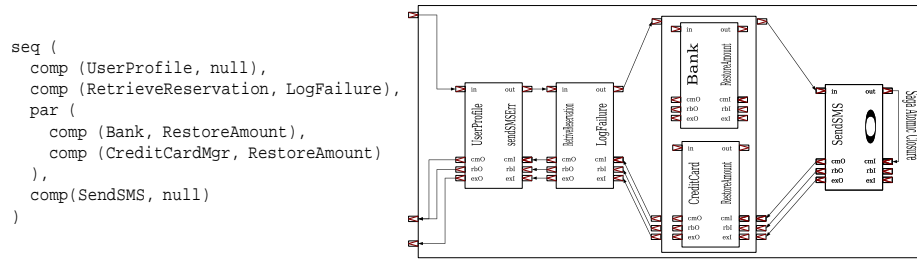


Figure 10. Stage3: The On-line Payment

payment of the fare are activated. This is done by the parallel execution of two activities: one performs the money transfer to the taxi company account (Bank service), the other charges the fare on the customer credit card (CrediCard service). In both cases, the compensations of failures restore the data on the corresponding account.

Our experimentation has shown that Naïve Sagas and JTWS provide a natural setting to design and deploy transactional business processes at a high level of abstraction. Indeed, the coordination details are hidden inside the JSCL implementation.

6 Concluding Remarks

Starting from a formal specification of parallel sagas we have presented JTWS, a Java API that provides the basic primitives for composing Web Services in (compensable, parallel) LRTs. The implementation is conceptually separated in two layers: JSCL and JTL. The former is a general framework for building networks of gates connected by typed signals. The latter is a specialised variant of JSCL where gates come equipped with few carefully selected signals that are tailored to the treatment of WSs transactions. The underlying JSCL layer makes the implementation fully distributed. The overall contribution is a setting for designing business process transactions where three level are reconciled: (1) a visual/graphical representation of parallel sagas, (2) a process calculus description in bijective correspondence with sagas diagrams, and (3) an executable, distributed translation of symbolic processes.

One interesting result of our experimentation is that level 2 is crucial for linking business analyst designs (level 1) to their actual implementations (level 3). Indeed, as already observed in [6], the process calculus formalisation forces us to deal with design choices that are not so evident at level 1 (e.g. centralised vs. distributed interrupt and compensation). Furthermore, level 3 can test and ensure that the design choices made at level 2 are really implementable / feasible.

As future work, we intend to exploit the flexibility of JSCL to implement and experiment with the alternative design choices identified in [6], including advanced features like nesting, speculative choice and alternative activities to provide a full-fledged transactional framework.

References

1. B. Anderson and D. Shasha. Persistent linda: Linda + transactions + query processing. In *Research Directions in High-Level Parallel Programming Languages*, pages 93–109. Springer-Verlag, 1992.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In E. Najm, U. Nestmann, and P. Stevens, editors, *Proc. of 6th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems (FMOODS'03)*, volume 2884 of *Lect. Notes in Comput. Sci.*, pages 124–138. Springer Verlag, 2003.
3. BPEL Specification (v.1.1). <http://www.ibm.com/developerworks/library/ws-bpel>.
4. WS BPEL issues list. http://www.choreology.com/external/WS_BPEL_issues_list.html.
5. Business Process Modeling Language (BPML). <http://www.bpml.org/BPML.htm>.
6. R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Reconciling two approaches to compensable flow composition. Submitted. Available at <http://www.di.unipi.it/~bruni/publications/concur2005-flow.ps.gz>.
7. R. Bruni, H. Melgratti, and U. Montanari. Nested commits for mobile calculi: extending Join. In J.-J. Lévy, E. Mayr, and J. Mitchell, editors, *Proceedings of the 3rd IFIP-TCS 2004, 3rd IFIP Intl. Conference on Theoretical Computer Science*, pages 569–582. Kluwer Academic Publishers, 2004.
8. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220. ACM Press, 2005.
9. R. Bruni and U. Montanari. Zero-safe net models for transactions in Linda. In U. Montanari and V. Sassone, editors, *Proceedings of ConCoord 2001, International Workshop on Concurrency and Coordination*, volume 54 of *Elect. Notes in Th. Comput. Sci.*, 2001.
10. N. Busi and G. Zavattaro. On the serializability of transactions in javaspaces. In U. Montanari and V. Sassone, editors, *Elect. Notes in Th. Comput. Sci.*, volume 54. Elsevier Science, 2001.
11. N. Busi and G. Zavattaro. On the serializability of transactions in shared dataspace with temporary data. In *Proceedings of ACM Symposium on Applied Computing (SAC'02)*, pages 359–366. ACM Press, 2002.
12. M. Butler, M. Chessell, C. Ferreira, C. Griffin, P. Henderson, and D. Vines. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.
13. M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In R. De Nicola, G. Ferrari, and G. Meredith, editors, *Proceedings of Coordination 2004*, volume 2949 of *Lect. Notes in Comput. Sci.*, pages 87–104. Springer Verlag, 2004.
14. M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In A. Abdallah and J. Sanders, editors, *Proceedings of 25 Years of CSP*, 2004.
15. T. Chothia and D. Duggan. An architecture for secure fault-tolerant global applications. *Theor. Comput. Sci.*, 322(3):567–613, 2004.
16. V. Danos and J. Krivine. Reversible communicating systems. In P. Gardner and N. Yoshida, editors, *Proceedings of CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004*, pages 293–307, 2004.
17. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
18. A. Hosking, S. Jagannathan, J. Vitek, and A. Welc. A semantic framework for designer transactions. In D. A. Schmidt, editor, *Proceedings of the 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *Lect. Notes in Comput. Sci.*, pages 249–263. Springer Verlag, 2004.

19. S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of COORDINATION 2004, Coordination Models and Languages, 6th International Conference*, volume 2949 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2004.
20. Java Transactional Web Services (JTWS). <http://www.di.unipi.it/~etuosto/jtws.html>.
21. C. Laneve and G. Zavattaro. Foundations of web transactions. In V. Sassone, editor, *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2005*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
22. B. Laskey and J. Parker. Microsoft biztalk server 2000: Building a reverse auction with biztalk orchestration, 2001. Microsoft Corporation. Available at <http://msdn.microsoft.com/library/en-us/dnbiz/html/bizorchestr.asp>.
23. F. Leymann. WSFL Specification (v.1.0). <http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, May 2001.
24. Oracle BPEL Process Manager. In <http://www.oracle.com/technology/bpel>.
25. D. Strollo. Composizionalità di transazioni e Web Services nell’ambito della telefonia mobile. Master’s thesis, Dipartimento di Informatica, Pisa, 2005. In Italian.
26. WebSphere Software Platform. <http://www-306.ibm.com/software/info1/websphere/index.jsp>.
27. Web Service Choreography Interface (WSCI) 1.0. <http://www.w3.org/TR/wsci>.
28. Web Services Conversation Language (WSCL) 1.0. <http://www.w3.org/TR/wscl10/>.
29. Web Services for Business Process Design (XLANG). http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.