Event-based Service Coordination *

Gianluigi Ferrari[1], Roberto Guanciale[2], Daniele Strollo[1,2] and Emilio Tuosto[3]

[1] Università degli Studi di Pisa, Dipartimento di Informatica
Largo B. Pontecorvo 3 I-56127, Pisa, Italy
{giangi,strollo}@di.unipi.it
[2] Institute for Advanced Studies IMT Lucca
Piazza S. Ponziano 6, 55100, Lucca, Italy
{roberto.guanciale,daniele.strollo}@imtlucca.it
[3] University of Leicester, Computer Science Department
University Road, LE17RH, Leicester, UK
et52@mcs.le.ac.uk

Abstract. In this paper we tackle the problem of designing and implementing a framework for programming service coordination policies. In particular, we illustrate the design and the prototype implementation of Java Signal Core Layer (JSCL), a coordination middleware for services based on the *event notification* paradigm. We formally motivate the design choices of the JSCL middleware by exploiting a variant of the π-calculus specifically tailored to deal with event notification and distribution. We demonstrate how service coordination polices can be precisely programmed in JSCL by some simple but illustrative case studies.

Keywords. Service Oriented Architectures, Event Notification, Coordination

1 Introduction

The web service protocol stack (e.g. WSDL, UDDI, SOAP) provides *basic* support for the development of service-oriented architectures by exploiting facilities to publish, discover and invoke network-available services. The service protocol stack has been extremely valuable to highlight the key innovative features of the service oriented computing approach. Most of the current development methodologies are focused on composition of services. Two different approaches can be adopted: *orchestration* and *choreography*. In the orchestration, services are thought as isolated and the main focus relies on their internal behavior. The participants have no acknowledgment of the surrounding network. An intermediate component, the *orchestrator* is responsible to arrange service activities according to the work-flow plan. This strategy provides a *local view* of the participants. From the other hand, the *choreography* model involves all parties

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project Sensoria and by the Italian FIRB Project Tocai.it.

and their associated interactions providing a *global view* of the system. Relevant standard technologies have emerged to model coordination policies. Among them, particular relevance is given to the Business Process Execution Language (BPEL4WS) [20], for the orchestration, and Web Service Choreography Description Language (WS-CDL) [28], for the choreography. However, it is not infrequent that such standards have drawbacks. In fact, constructs are often informally specified which usually leads to ambiguities or redundancy. Several research and implementation efforts are currently devoted to provide a clear semantics for their constructs and tools for verification (COWS [21], Global Calculus [8], λ_{req} [1] ORC [24], SCC [3], SOCK [18] to cite a few). However, research is still underway.

A well known paradigm for specifying and programming distributed systems is the *event notification* paradigm (EN, for short), where distributed computational components can act as publishers and/or subscribers. When a component intends to send data to or requests a service from other components, it issues an event that eventually shall trigger a reaction from subscribers that previously subscribed for such kind of events. The EN paradigm seems to provide a suitable framework to deal with *service oriented architectures* (SOAs) that require components to be loosely coupled. Specifically, the EN paradigm features high level coordination mechanisms that allow programmers/designers to decouple components and rely entirely on event handling.

In this paper, we report our experience in using the EN paradigm to design and implement coordination policies for SOA. We designed and implemented a middleware called Java Signal Core Layer [12] (JSCL). A distinguished feature of JSCL consists in the strict interplay among formal semantic foundations, implementation pragmatics and experimental evaluation of the resulting programming constructs. More precisely, all the programming facilities available in JSCL have a clear semantics. Indeed, at the abstract level, the middleware takes the form of the *Signal Calculus* [12,13,11] (`SC`). The `SC` is a variant of the π-calculus [26] with explicit primitives to deal with event notification and component distribution. At the implementation level, JSCL takes the form of a collection of Java API equipped with a standard development environment (an Eclipse plug-in). The JSCL API's are available at `www.tao4ws.net`.

The `SC` allows one to define services coordination policies (orchestration and choreography) relying on event notification only. Moreover, it features sessions as a mechanism to synchronize work-flows of distributed and independent components. Remarkably, `SC` does not assume any centralized mechanism for publishing, subscribing and notifying events. Indeed, each subscriber explicitly defines the class of events it is interested in. In [22] this pattern is referred to as *non brokered*, in contrast with the *brokered* solutions that implements publish/subscribe mechanisms on top of a classification of signals without taking into account the involved components. Basically, brokered solutions rely on global state space e.g. *linda tuple spaces* [15].

All `SC` notions are reflected in the JSCL API's. Indeed, the design choices underlying the JSCL implementation have been formally motivated in terms of

the SC. Hence, SC and JSCL can be regarded as a foundational framework and its programming counterpart for specifying, verifying and programming coordination policies of distributed services.

We envisage the impact of our approach on the service oriented computing technologies as follows. Conceptually, the JSCL-SC framework adds a further layer to the basic web service protocol stack (SOAP, UDDI, WSDL). The JSCL-SC layer provides the formal and programming mechanisms to design, verify and program web service coordination policies (e.g. a BPEL4WS orchestrator or a WS-CDL choreography) on top of the basic service protocols. Figure 1 pictorially illustrates the context of our approach.

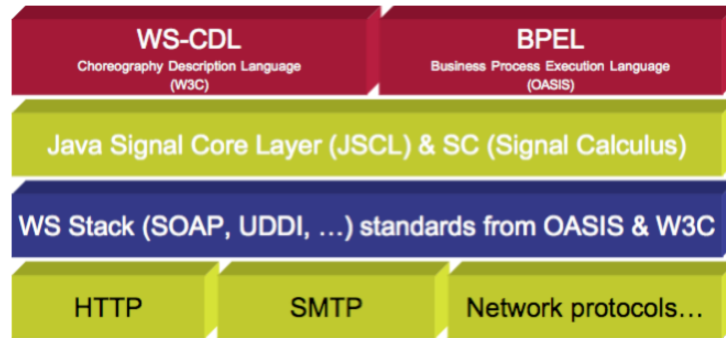

Fig. 1. JSCL-SC technological context

In this paper we outline the main features of our framework. We refer to [12,13,17,27] for the details. We demonstrate the usefulness of JSCL in the practical programming of coordination policies by some case studies. The focus of our experiments has been on the design and implementation of the work-flow of the coordination, taking into account the possibility of handling long-running transactions in the style of SAGA compensations [14,5].

This paper is organized as follows. In Section 2 we review the SC process calculus. Section 3 outlines the architectures of the JSCL middleware. Section 4 discusses the case studies. Section 5 yields some concluding remarks.

2 Signal Calculus

In this section, we present a simplified version of SC focusing on session managing only. We assume a set of *topic* names (ranged over by τ), a set of signal variables (ranged over by x) and a set of signal names (ranged over by $s, s_1, s_2 ...$). In the following the name n ranges over signal names and signal variables. Signal names represent data exchanged among components and should carry additional information even if this feature is not explicitly modeled here. Finally, we assume

$$
\begin{array}{lll}
B ::= & \mathbf{out}\langle n : \tau \copyright \tau' \rangle.B & \textit{(Signal emission)} \\
| & (\nu\tau)B & \textit{(Topic restriction)} \\
| & \mathbf{rupd}\,(R)\,.B & \textit{(Reaction update)} \\
| & \mathbf{fupd}(F).B & \textit{(Flow update)} \\
| & B \mid B' & \textit{(Parallel)} \\
| & !B & \textit{(Bang)} \\
| & 0 & \textit{(Empty behavior)}
\end{array}
$$

Fig. 2. Behaviors

a set of component names $a, b, \ldots$. Hereafter, we adopt the notation $\vec{a}$ to denote a set of component names.

The calculus is centered around the notion of *component*, written as $a[B]_F^R$, that represents a service uniquely identified by a name a, the public address of the service, with internal behavior B and interfaces R and F respectively called *reactions* and *flows*.

Figure 2 displays the syntax of SC behaviors. The *signal emission* $\mathbf{out}\langle n : \tau \copyright \tau' \rangle.B$ spawns signal n of topic τ over the session τ' and then continues as B. Topics can be generated dynamically by *restriction*. Restriction acts as a binder for the declared topic; namely, the occurrences of τ in $(\nu\tau)B$ are bound. The calculus provides two primitives to allow a component to dynamically change its interface: the *reaction update* $\mathbf{rupd}\,(R)\,.B$ and the *flow update* $\mathbf{fupd}(F).B$. The former installs a new reaction R in the interface part of components and the latter adds F to its flows. Recursive behaviors are obtained via replication $!B$ with the usual interpretation that there are infinitely many copy of B available, namely $!B$ is equivalent to $B \mid !B$. The empty and parallel constructs have the obvious meaning.

Reactions describe how a component reacts upon the reception of a signal and their syntax is given by the following grammar:

$$R ::= 0 \quad | \quad x : \tau \copyright \lambda\tau' \rightarrow B \quad | \quad x : \tau \copyright \tau' \rightarrow B \quad | \quad R|R$$

The *empty reaction* 0 cannot react to any signal. The *lambda reaction* $x : \tau \copyright \lambda\tau' \rightarrow B$ is triggered by signals having topic τ independently from the actual session the event belongs to. The *check reaction* $x : \tau \copyright \tau' \rightarrow B$ reacts only to signals having topic τ issued for the session identified by the topic τ'. Once a reaction has been fired, the behavior B is spawned in the component and will start its executed in parallel with the existing behaviors. Notice that for a lambda reaction the name τ' is bound in the behavior B, while for a check reaction it is a free name. Moreover, in both reactions the variable x acts as a binder for the name of the received signal. *Reaction composition* allows a component to react to different kinds of signal in different ways.

Flows describe the component view of the coordination policies. Their syntax is defined as follows:

$$F ::= 0 \quad | \quad \tau \rightsquigarrow \vec{a} \quad | \quad F|F.$$

The *empty flow* 0 does not deliver any kind of signals, the *single flow* $\tau \rightsquigarrow \vec{a}$ delivers signals having topic τ to the components specified in the set $\vec{a}$ and, finally, flows can be composed in parallel.

Networks describe the component distribution and carry signals exchanged among components (the syntax of networks is given below).

$$N ::= \emptyset \quad | \quad a[B]_F^R \quad | \quad N \parallel N \quad | \quad \langle s : \tau © \tau' \rangle @a \quad | \quad (\nu\tau)N$$

A network can be empty $\emptyset$, a single component $a[B]_F^R$, or the parallel composition of networks $N \parallel N'$. Networks handle signals exchanged among components. The signal emission spawns into the network, for each target component, an "envelope" $\langle s : \tau © \tau' \rangle @a$ containing the signal and the target component name a. Finally, restriction allows the scope extrusion of freshly generated topics over networks.

2.1 Operational Semantics

We now present the operational semantics of the SC.

The structural congruence over reactions, flows, behaviors and networks is the smallest congruence relation that satisfies the commutative monoidal laws for $(R, |, 0)$, $(F, |, 0)$, $(B, \ | \ , 0)$ and $(N, \|, \emptyset)$. Additionally, the following laws hold:

$$\begin{array}{ll} (\nu\tau)0 \equiv 0, & ((\nu\tau)B) \mid B' \equiv (\nu\tau)(B \mid B'), \text{ if } \tau \notin fn(B') \\ (\nu\tau)\emptyset \equiv \emptyset, & ((\nu\tau)N) \parallel N' \equiv (\nu\tau)(N \parallel N'), \text{ if } \tau \notin fn(N') \end{array}$$

and, if $B \equiv B'$,

$$x : \tau © \lambda\tau' \rightarrow B \equiv x : \tau © \lambda\tau' \rightarrow B' \quad (1)$$

$$x : \tau © \tau' \rightarrow B \equiv x : \tau © \tau' \rightarrow B' \quad (2)$$

where x can be alpha converted both in (1) and (2), while τ' only in (1).

Finally, for the structural congruence over networks the following equations hold:

$$a[0]_F^0 \equiv \emptyset, \qquad \frac{F_1 \equiv F_2 \quad B_1 \equiv B_2 \quad R_1 \equiv R_2}{a[B_1]_{F_1}^{R_1} \equiv a[B_2]_{F_2}^{R_2}}, \qquad \frac{\tau \notin fn(R) \cup fn(F) \cup \{a\}}{a[(\nu\tau)B]_F^R \equiv (\nu\tau)a[B]_F^R}.$$

To simplify the definition of the reduction relation over networks, we introduce an auxiliary function on flows. The *flow projection*, $(F)\downarrow_\tau$, is inductively defined as follows:

$$(\tau \rightsquigarrow \vec{a})\downarrow_\tau = \vec{a} \qquad (\tau \rightsquigarrow \vec{a})\downarrow_{\tau'} = (0)\downarrow_{\tau'} = \emptyset \qquad (F_1|F_2)\downarrow_\tau = (F_1)\downarrow_\tau \cup (F_2)\downarrow_\tau$$

Intuitively, the projection $(F)\downarrow_\tau$ takes a flow and a topic and yields the set of names of components which have subscribed for events of topic τ. In other words, when a signal having topic τ occurred, it must be delivered to all the subscribed components, identified by the names in the set $(F)\downarrow_\tau$.

$$a[\mathbf{rupd}\,(R')\,.B' \mid B]_F^R \to a[B' \mid B]_F^{R|R'} \quad \text{(ReactionUpd)}$$

$$a[\mathbf{fupd}(F').B' \mid B]_F^R \to a[B' \mid B]_{F|F'}^R \quad \text{(FlowUpd)}$$

$$\frac{(F)\downarrow_\tau = \{b_1, \ldots, b_n\}}{a[\mathbf{out}\langle s : \tau\copyright\tau'\rangle.B' \mid B]_F^R \to a[B' \mid B]_F^R \parallel \langle s : \tau\copyright\tau'\rangle @b_1 \parallel \ldots \parallel \langle s : \tau\copyright\tau'\rangle @b_n} \text{(Emit)}$$

$$\langle s : \tau\copyright\tau'\rangle @a \parallel a[B]_F^{x:\tau\copyright\tau' \to B'|R} \to a[B|\{s/x\}B']_F^R \quad \text{(RCActivation)}$$

$$\langle s : \tau\copyright\tau'\rangle @a \parallel a[B]_F^{x:\tau\copyright\lambda\tau_1 \to B'|R} \to a[B|\{s/x\}\{\tau'/\tau_1\}B']_F^{x:\tau\copyright\lambda\tau_1 \to B'|R} \quad \text{(RLActivation)}$$

$$\frac{N \to N'}{N \parallel N_1 \to N' \parallel N_1} \text{(NStep)} \qquad \frac{N \equiv N' \quad N \to N_1}{N' \to N_1} \text{(NStruct)}$$

Fig. 3. Operational semantics

The reduction relation $\to$ over networks is defined by the rules depicted in Figure 3. Reactions can be added to a component by the rule *ReactionUpd*. The rule extends the interface of the component named a by appending to the set of installed reactions the new reaction. Similarly, the *FlowUpd* extends the flow interface of a component by appending the new flow. The *Emit*, *RCActivation* and *RLActivation* rules define notification dispatching: at emission time, component a spawns into the network a signal targeted to all the components ($c_i \in \vec{b}$). Once a signal envelope has been spawn into the network, the *RCActivation* or the *RLActivation* rules can be applied in accordance with the kind of the installed reactions. Notice that the application of these rules activates the behavior associated to the reactions applying the suitable variable substitutions. Finally a check reaction has a *stateless* interpretation: after its execution, the reaction is removed by the component interface.

2.2 Joining events

To explain the SC programming model, we specify a work-flow synchronization mechanism. Let us consider a network consisting of four components: an emitter E, two intermediate components C_1 and C_2, and the join service J. The emitter E starts the communications raising toward C_1 and C_2 two events having different topics. Both components C_1 and C_2 perform an internal computation and then notify their termination by issuing an event to the join service J. The join service J waits for the termination of both components and then executes its internal behavior B. The signals sent to C_1 and C_2 are both related to the same session τ. This session is later used by the component J to synchronize the work-flow. The two intermediate services C_1 and C_2 can concurrently perform

their tasks, while the execution of the service J can be triggered only after the completion of their executions.

This work-flow pattern can be specified by the `SC` network $E \parallel C_1 \parallel C_2 \parallel J$, where:

$$E \triangleq e[(\nu\tau)\mathbf{out}\langle s:\tau_1 © \tau\rangle.\mathbf{out}\langle s:\tau_2 © \tau\rangle.0]^{0}_{\tau_1 \rightsquigarrow c_1 | \tau_2 \rightsquigarrow c_2}$$
$$C_i \triangleq c_i[0]^{x:\tau_i © \lambda\tau \rightarrow \mathbf{out}\langle x:\tau_i © \tau\rangle.0}_{\tau_i \rightsquigarrow j}, \qquad i = 1, 2$$
$$J \triangleq j[0]^{x:\tau_1 © \lambda\tau \rightarrow \mathbf{rupd}\left(x':\tau_2 © \tau \rightarrow B\right).0}_{0}$$

The join component has only one active reaction installed for signals having topic τ_1. When the two intermediary services forward their signals, the envelope containing the τ_2 event cannot be consumed by the join, and remains pending over the network. The reception of the τ_1 envelope triggers the activation of the join generic reaction. The reaction *reads* the session of the signal τ_1 and creates a new specialized reaction for the signal topic τ_2. This reaction can be triggered only by signals that refer to the session received by the τ_1 signal. Once such kind of signal is received, the behavior B is executed.

3 Java Signal Core Layer

The Signal Calculus has been used to formally drive the prototype implementation of a middleware, called Java Signal Core Layer (JSCL), aimed at programming service coordination policies in an event notification based paradigm.

JSCL has been designed and implemented by a two-level architecture reflecting the structure of the `SC`. The lower level is called the *Inter Object Communication Layer* (`iocl`). The `iocl` layer provides the primitives for handling network interactions. Indeed, the `iocl` layer abstracts from the actual networking technologies in order to hide the network complexity to the higher layer called *Signal and Component Layer* (SCL). The naming facilities for identifying components, the capabilities for data serialization and message delivering have been implemented by the `iocl` layer. Several instances of the `iocl` may coexist within a JSCL program. The basic idea is that each `iocl` instance acts as the bridge among several network infrastructures (e.g. Web Services, CORBA, remote methods, etc). Intuitively, the `iocl` represents the `SC` *network* and supports component distribution and the notification/delivery of messages to the distributed components. The SCL layer provides all the facilities to create and handle components, signals, reactions by introducing suitable Java API's. Hence, JSCL supports implementation of event-based coordination policies by enabling Java-like programming techniques.

We do not discuss here the concrete implementation of the JSCL layers (we refer to [27] for the detailed presentation). In order to give a flavor of how JSCL has been implemented, we present the JSCL signal delivery protocol. This protocol is displayed in Figure 4. Once the component $S1$ raises a new event, the resulting notification is delivered to the list of components ($S2$) subscribed for the

corresponding signal type. The signal delivering is implemented by demanding the local `iocl` service $iocl_L$ to serialize the message and to contact the remote `iocl` service $iocl_R$ (Step 1). This step is performed in an asynchronous way (Step 2). Once the message has been received, the remote `iocl` service $iocl_R$ performs the data deserialization and forwards the signal to the component $S2$ (Step 3).

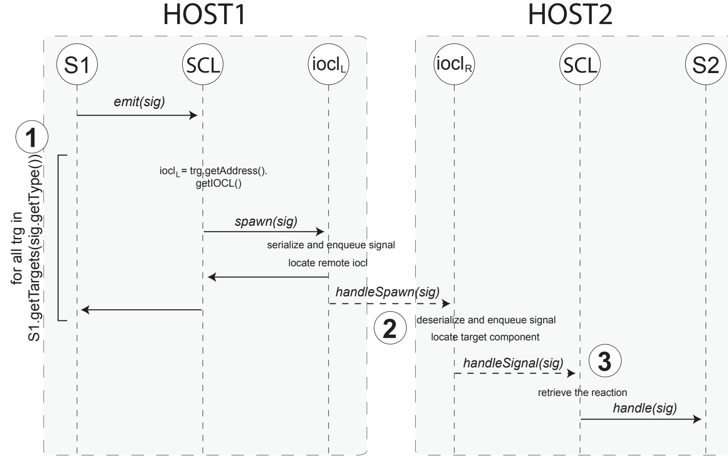

Fig. 4. Signal delivery protocol

4 Case Studies

To illustrate some of the features and the design-programming facilities made available by our framework, we consider two small case studies. First, we address the problem of composing Web Services in long-running transactional business processes, where compensations must be dealt with appropriately. We illustrate the JSCL implementation of a module which provides suitable primitives for wrapping and invoking Web Services as activities in long-running transactions. The second case study concerns the design and implementation of a car emergency system. We assume a car equipped with a diagnostic system that continuously reports on the status of the vehicle. When the car experiences some major failure (e.g. engine overheating, exhausted battery, flat tires) the in-car emergency service is invoked to select the appropriate tow-truck and garage services.

4.1 Implementing Long Running Transactions in JSCL

One of the emerging issues when aggregating Web Services is constituted by the so-called *long-running transactions* (LRTs), i.e., the possibility of requiring a set

of Web Services 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 loosely coupled and any of them is responsible for maintaining the consistency on local data. In order to achieve atomicity, LRTs may use *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 standards proposed for orchestrating Web Services (e.g. BPEL4WS [20]) include primitives for handling LRTs. Noteworthy, all those proposals formalize the orchestration syntax but not the semantics, whose informal description can make the intended behavior of constructs ambiguous and can lead to different implementations of the same language.

Here, we take advantage of a formal framework for isolating and studying LRTs since our goal is to build a framework for coordinating transactional compositions over a solid formal basis. The formal framework we choose is Naïve Sagas [5], a process calculus for compensable transactions. From the existing calculi for LRTs [6,7,10,2,9,19,23], we have chosen Naïve Sagas because it exposes the orchestration mechanism behind LTRs. In fact, activities in a saga are described at the high level of abstraction, where the elementary actions are not interpreted. Transactional flows 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 for 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 for A, and (*ii*) the control is forwardly propagated to the other stages of the transactions. In case of a failure in the rest of the transaction, the backward flow starts so that the effects of executing A must be rolled back. This is achieved by activating the installed compensation B and afterward by propagating the rollback to the activities that were executed before A. Note that B is not installed if A is not executed.

With JSCL the transactional blocks are obtained by suitable wrappers, called *Transactional Components* (TC). To implement the behavior of a transactional component we need three kinds of topics

- SIG_{CMT} is used to notify that the entire work-flow has been successfully completed,
- SIG_{FW} is used to activate the next steps of the chain within a transactional work-flow (forward flow),
- SIG_{RB} is used to activate the compensations (backward flow).

The JSCL implementation of the transactional component (TC) is illustrated in Code 1 in the appendix. A TC is constructed (see lines 3-7) by specifying its address, the main activity (A) to be performed and its compensation (C). The component is initialized by creating the flows for both the SIG_{RB} and SIG_{FW} topics (see lines 13-19). The declaration of reactions for TC is implemented by the method `initReactions`. Notice that TC installs just one reaction for handling requests of SIG_{FW} (lines 23-59). The line 27 declares the activation

condition for the installed reaction. The first parameter, FW, declares the topic, while the second parameter is used to specialize reactions on a session (*null* is used for *lambda reactions*).

The block 28-58 contains the declaration of the task to be executed. Once a signal having topic SIG_{FW} is received, the method `handle` is invoked. The components tries to execute the main activity A (line 33) and if a failure happens, the rollback signal is sent out (lines 37-38) and the reaction terminates. If A is successfully executed, the compensation for that session flow can be installed (block 45-55). Notice that, in this case the rollback will be activated only for the requests owning to the current session. A check reaction is used to handle this case. The reaction for rollback requests consists of executing the compensation activity C propagating the request along the backward chain.

The JSCL implementation of Naïve Sagas provides components implementing *parallel* and *sequential* structural composition of *transactional gates*. The composition constructs keep the structure of TC and can be reused in further compositions. Figure 5 shows the (intuitive) implementation of the Naïve Sagas compensable process $P \triangleq A_1 \div B_1; A_2 \div B_2; A_3 \div B_3$.

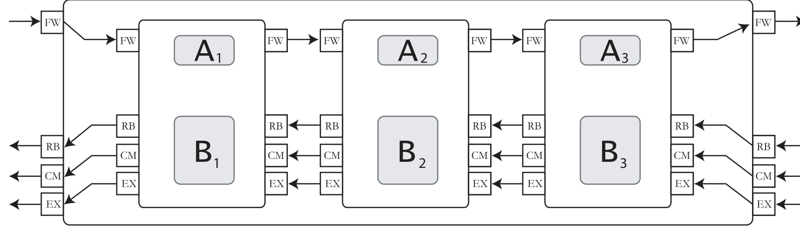

Fig. 5. SAGA compensable process: an example.

Correctness of the JSCL implementation of Naïve Sagas can be formally stated via a semantic preserving encoding of Naïve Sagas in SC (we refer to [17] for the technical treatment). Here, we simply provide via an example the intuition of the encoding. We assume as given two functions $[\![A \div B]\!](x, \tau_s)$ and $[\![B]\!]_{rb}(x, \tau_s)$ that translate a Naïve Sagas process $A \div B$ and the compensation B to SC internal behaviors. The two functions work on signal named x having session τ_s. We also assume that the first function translates the successful return statements into the signal emission $\mathbf{out}\langle x : fw©\tau_s\rangle.\mathbf{rupd}\,(x : rb©\tau_s \rightarrow [\![B]\!]_{rb}(x, \tau_s))$ and the exception rising into $\mathbf{out}\langle x : rb©\tau_s\rangle.0$, and that the second function translates the successful return statements into the signal emission $\mathbf{out}\langle x : rb©\tau_s\rangle.0$ and the exception rising into $\mathbf{out}\langle x : ex©\tau_s\rangle.0$. The Naïve Sagas compensable

process, previously described, is represented by the SC network $[\![P]\!]$:

$$[\![A_1 \div B_1]\!] \triangleq p_1[0]^{x:fw ©\lambda\tau_s \to [\![A_1 \div B_1]\!](x,\tau_s)}_{fw \rightsquigarrow p_2}$$

$$[\![A_2 \div B_2]\!] \triangleq p_2[0]^{x:fw ©\lambda\tau_s \to [\![A_2 \div B_2]\!](x,\tau_s)}_{fw \rightsquigarrow p_3 | rb \rightsquigarrow p_1}$$

$$[\![A_3 \div B_3]\!] \triangleq p_3[0]^{x:fw ©\lambda\tau_s \to [\![A_3 \div B_3]\!](x,\tau_s)}_{rb \rightsquigarrow p_2}$$

$$[\![P]\!] \triangleq [\![A_1 \div B_1]\!] \mid [\![A_2 \div B_2]\!] \mid [\![A_3 \div B_3]\!]$$

Conceptually, our implementation of Naïve Sagas compensable processes add a further layer to JSCL. This new layer exploits JSCL primitives to define the behavior of transactional constructs according to Naïve Sagas. In other words, Naïve Sagas compensable processes become a specialized variant of JSCL where gates come equipped with few carefully selected signals that are tailored to the treatment of web service transactions. The underlying JSCL layer makes the implementation fully distributed. The prototype implementation of Naïve Sagas compensable process in JSCL has been first presented in [4]. We refer to [27] for a more detailed treatment of the implementation issues.

4.2 The Car Emergency System

In this section we illustrate how the SC can be used for modeling the service coordination issues of the SENSORIA car emergency system [29] where a car manufacturer provides an assistance service to its customers. Once a customer's car breaks down, the system attempts to locate a garage, a tow truck and a rental car service so that the car is towed to the garage and repaired meanwhile the customer may continue his travel. The inter-dependencies between the booking services are summarized as follows:

– the first step is to charge the credit card with a security amount;
– before looking for a tow truck, a garage must be found as it poses additional constraints to the candidate tow trucks;
– if finding a tow truck fails, the garage appointment must be revoked;
– if renting a car succeeds and finding either a tow truck or a garage appointment fails, the car rental must be redirected to the broken down car's actual location;
– if the car rental fails, it should not affect the other services.

To describe the work-flow and the inter-dependencies among services we exploit the standard Business Process Modeling Notation (BPMN [16]) (see Figure 6). Notice that the specification above exploits the transactional and compensation facilities of BPMN.

The Car Emergency System: The SC specification We now formally describe the Car Emergency System in SC. Each participant is represented by a SC component making use of the three types of signals below:

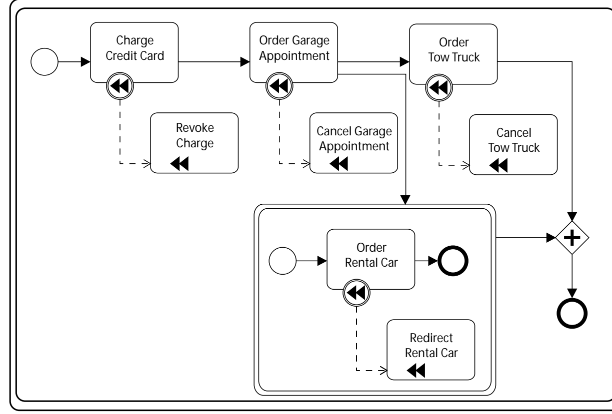

Fig. 6. Car emergency system: the BPMN Specification

– τ_f is used by *forward signals*. These signal propagate the information on the completion of previous activities in the work-flow
– τ_r is used by *rollback signals*. Rollback signals are backwardly propagated by components when their compensations have been executed;
– τ_n is used to notify the current work-flow session to components that have to synchronize several work-flow paths.

A component that represents a BPMN activity is a *transactional component* and can instantiate either a reaction that handles τ_f signals or one that handles τ_r signals. We now specify the basic building block of the SC specification transactional component $TC = TC(a, A, C, \overrightarrow{prev}, \overrightarrow{next})$ where a denotes the location of the component, A the internal behavior and C the compensation. We assume that it propagates τ_f signals to the components in $\overrightarrow{next}$ and τ_r signals to the components in $\overrightarrow{prev}$. The boolean parameter *sub* states if the component is a sub-transaction. In the following, for the sake of readability, we use *prev* and *next* to denote the sets $\overrightarrow{prev}$ and $\overrightarrow{next}$, respectively. Hereafter, we also assume that:

1. if A successfully terminates, then a τ_{ok} signal issued;
2. if A fails, then an exception (a τ_{exc} signal) is raised to inform the component to start the backward flow

$$TC(a, A, C, prev, next, sub) \triangleq a[0]^{R_{TC}(A,C,sub)}_{\tau_f \rightsquigarrow next | \tau_r \rightsquigarrow prev | \tau_{exc} \rightsquigarrow a | \tau_{ok} \rightsquigarrow a}$$

where:

$$R_{TC}(A,C,sub) \triangleq \begin{array}{l} x:\tau_f \copyright \lambda\tau \rightarrow \\ \quad \mathbf{rupd}\left(\begin{array}{l} x:\tau_{ok}\copyright\tau \rightarrow \\ \quad \mathbf{rupd}\left(\begin{array}{l} x':\tau_r\copyright\tau \rightarrow \\ \quad C. \\ \quad \mathbf{out}\langle x':\tau_r\copyright\tau\rangle. \\ \quad 0 \end{array}\right). \\ \quad \mathbf{out}\langle x:\tau_f\copyright\tau\rangle.0 \\ x:\tau_{exc}\copyright\tau \rightarrow B_{exc}(x,\tau,sub) \end{array}\right) \mid \\ \quad A \end{array}$$

and:

$$B_{exc}(x,\tau,sub) \triangleq \begin{cases} \mathbf{out}\langle x:\tau_r\copyright\tau\rangle.0 & if\ sub = false \\ \mathbf{rupd}\left(\begin{array}{l} x':\tau_r\copyright\tau \rightarrow \\ \quad \mathbf{out}\langle x':\tau_r\copyright\tau\rangle. \\ \quad 0 \end{array}\right). & otherwise \\ \mathbf{out}\langle x:\tau_f\copyright\tau\rangle.0 & \end{cases}$$

In the initial state, the component TC can react only to a τ_f signal. Hence, the session τ starts and A is executed. Concurrently with the activity A, the component installs the reactions to check termination of A (i.e. the emission of signal τ_{ok} or signal τ_{exc}). If the execution of A is successful, a check reaction for a further rollback notification (τ_r) is installed, and the τ_f signal is propagated to the successive stages in the work-flow ($\mathbf{out}\langle x:\tau_f\copyright\tau\rangle.0$). When a τ_r signal for session τ is received, the compensation C is executed and the rollback signal is propagated to previous stages ($\mathbf{out}\langle x':\tau_r\copyright\tau\rangle.0$). Notice that two handlers (B_{exc}) for the exception of the main activity are provided, according to the *sub* parameter. In the first case, the handler simply starts the backward flow, raising a rollback signal. In the second case, the handler propagates the τ_f signal, since an error of the sub-transaction should not affect the computation of the other components. Moreover the component installs a reaction for τ_r to forward backward flow without executing the compensation.

A sequential work-flow is specified as a chain of transactional components by setting the *next* and *prev* sets suitably. To model the parallel branch, we define the *collector* and *emitter* components as follows:

$$Emitter(a,prev,next,collector) \triangleq$$
$$a[0]^{x:\tau_f\copyright\lambda\tau\rightarrow\mathbf{rupd}(x':\tau_r\copyright\tau\rightarrow\mathbf{rupd}(x'':\tau_r\copyright\tau\rightarrow\mathbf{out}\langle x'':\tau_r\copyright\tau\rangle.0)).\mathbf{out}\langle x:\tau_n\copyright\tau\rangle.\mathbf{out}\langle x:\tau_f\copyright\tau\rangle.0}_{\tau_f\rightsquigarrow next|\tau_r\rightsquigarrow prev|\tau_n\rightsquigarrow\{collector\}}$$

$$Collector(a,prev,next) \triangleq$$
$$a[0]^{x:\tau_n\copyright\lambda\tau\rightarrow\mathbf{rupd}(x':\tau_f\copyright\tau\rightarrow\mathbf{rupd}(x'':\tau_f\copyright\tau\rightarrow\mathbf{rupd}(x''':\tau_r\copyright\tau\rightarrow\mathbf{out}\langle x''':\tau_r\copyright\tau\rangle.0.\mathbf{out}\langle x'':\tau_f\copyright\tau\rangle.0)))}_{\tau_f\rightsquigarrow next|\tau_r\rightsquigarrow prev}$$

The emitter represents the entry point of the parallel branch. Basically, it activates the forward flow of *next* components, and synchronizes their backward

flows. The synchronization mechanism is implemented by installing two reactions for the topic τ_r and the session τ (through $\mathbf{rupd}\,(x' : \tau_r©\tau \rightarrow \mathbf{rupd}\,(x'' : \tau_r©\tau \rightarrow ...)))$. After that the synchronization mechanism has been installed, the emitter activates the forward flow ($\mathbf{out}\langle x : \tau_n©\tau\rangle.\mathbf{out}\langle x : \tau_f©\tau\rangle.0$). Notice that the component emits two signals: one having topic τ_f and the other one having topic τ_n. The first signal is delivered to the components representing the parallel activities. The other one is delivered to the collector, informing it of the received session that will be later used by it to implement its synchronization. When the synchronization of the backward flow takes place, the emitter forwards the rollback signal ($\mathbf{out}\langle x'' : \tau_r©\tau\rangle.0$) to the *prev* components.

Similarly, the collector component is responsible to implement the synchronization mechanism for the forward flows and to activate the backward flows of the parallel components when a τ_r signal is received. Notice that the collector exploits a τ_n signal to get information about the session τ.

Summing up, the car emergency system is specified by the following `SC` network:

$$\begin{array}{l}
TC(card, ChargeCredit, RevokeCredit, \{\}, \{garage\}) \parallel \\
TC(garage, OrderGarage, CancelGarage, \{card\}, \{e\}) \parallel \\
Emitter(e, \{garage\}, \{truck, car\}, \{c\}) \parallel \\
TC(truck, OrderTowTruck, CancelTowTruck, \{e, car\}, \{c\}) \parallel \\
TC(car, OrderCar, RedirectCar, \{e\}, \{c\}) \parallel \\
Collector(c, \{truck, car\}, \{\})
\end{array}$$

The Car Emergency System: the JSCL implementation The JSCL middleware come equipped with an environment in the form of an Eclipse plug-in that supports the design and development of service coordination policies.The JSCL development environment is composed by three layers: an editor that permits to graphically model service coordination policies, a model transformation that compiles a model into Java code and the JSCL middleware as runtime support. Our development methodology consists of three steps.The first step consists of the graphical definition of the coordination policy of services. The graphical model obtained is then used as input of a compilation facility that generates the JSCL code where the internal logic of each service is rendered through suitable annotations. Finally, the annotations must be finalized to implement the internal behavior of each service.

The JSCL graphical notation captures the sequence of activities via the description of the network topology of components involved in the work-flow. This notation has some similarities with BPMN. The main difference is that BPMN defines directly the flow of the messages exchanged among components exploiting a standard flow-chart notation. Our notation defines the correlation among services, while the message sequence depends by the component internal behavior and is not directly caught at design time. Figure 7 provides the snapshot of the design of the Car Emergency System within the JSCL environment.

Notice that our design exploits a specialized JSCL AndComponent (a generalization of the Join component introduced in Section 2). The first AndCom-

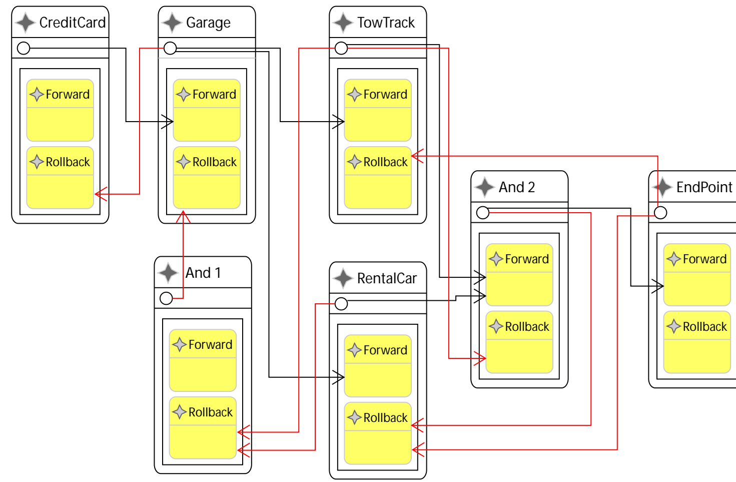

Fig. 7. the JSCL Graphical Design

ponent is used to synchronize the backward flow before the Garage compensation. The second AndComponent has a twofold role: (*i*) to synchronize the forward flow, and (*ii*) to execute the compensation of RentalCar if both the OrderTowTruck fails and the RentalCar main activity has been completed successfully. Notice also that the OrderTowTruck compensation is not executed if the RentalCar fails. The EndPoint component represents the BPMN final state. This component simply forwards the received signals without change their type.

The graphical model is then used to generate the *template* JSCL code where the internal logic of each component has not yet been implemented. Below we show the template code of the ForwardTruck.

```
protected class ForwardTruck extends SignalHandlerTask {
    public Object handle ( Signal  s ){
       try {
          TruckComponent parent = (TruckComponent) getParent();
          //  Program here the internal logic
          parent.state.set(s,ID(), true) ;
          emit(s) ;
        }
        catch (Exception e ) {
          s.setType (ROLLBACK) ;
          emit ( s ) ;
        }
   }
}
```

Once the template code has been generated, it is possible to implement the component internal logic using standard Java programming techniques.

5 Concluding remarks

The SC-JSCL framework has been design to support the specification, the implementation and verification of coordination policies for services oriented applications. Our main goal is to provide general facilities to implement high-level languages for service oriented architectures (e.g. BPEL4WS [20], BPML [25],WS-CDL [28]). The strict interplay between SC and JSCL permits to drive and verify implementation of such languages.

A number of approaches have been introduced to provide the formal foundations of standards for service orchestrations and service choreographies. The SC-JSCL framework differs from these approaches (COWS [21], Global Calculus [8], λ_{req} [1] ORC [24], SCC [3], SOCK [18] to cite a few), since it focus on a lower level of abstraction, merging the theoretical formalization with the implementation requirements. Indeed, the emphasis in SC-JSCL is just on designing general facilities to program coordination patterns on services by exploiting the notion of event notification.

There are a number of directions that we are pursuing for the future development of our framework. In [13], we introduced an algebraic structure over topics. This allows us to implement complex coordination logics directly inside the signal type. Moreover, this provides the foundational description of BPMN-like gateways. We intend to investigate this issue in order to design a BPMN work-flow engine based on the SC/JSCL framework. Furthermore, we plan to extend the SC/JSCL framework with facilities for reasoning and proving properties of coordination policies. On one hand, we are extending the compilation facilities so to generate both the source JSCL code and the SC specification out of the JSCL graphical notation. On the other hand, we plan to integrate in our environment toolkits that provide verification and analysis capabilities for Java programs and other semantic checker (e.g. bisimulation and model checkers) for the SC specification.

References

1. M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino. Secure service orchestration. In *FOSAD*, volume 4667 of *Lecture Notes in Computer Science*. Springer, 2007.
2. Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro. A calculus for long-running transactions. In *Proceedings of FMOODS 2003*, volume 2884 of *Lect. Notes in Comput. Sci.*, pages 124–138, 2003.
3. Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. Scc: A service centered calculus. In *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.
4. Roberto Bruni, Gianluigi Ferrari, Hernán Melgratti, Ugo Montanari, Daniele Strollo, and Emilio Tuosto. Java Transactional Web Services: from theory to practice in compensable, distributed long-running transactions. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *Proceedings of the 2nd International*

Workshop on Web Services and Formal Methods, volume 3670 of *Lect. Notes in Comput. Sci.*, pages 272–286. Springer-Verlag, 2005.
5. Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *Annual Symposium on Principles of Programming Languages POPL*, pages 209–220. ACM Press, 2005.
6. Michael Butler and Carla Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Proceedings of Coordination 2004*, volume 2849 of *Lecture Notes in Computer Science*, pages 87–104, 2004.
7. Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Proc. of 25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150, 2005.
8. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.
9. T. Chothia and D. Duggan. An architecture for secure fault-tolerant global applications. *Theor. Comput. Sci.*, 322(3):567–613, 2004.
10. Vincent Danos and Jean Krivine. Reversible communicating systems. In *Proceedings of CONCUR 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 293–307, 2004.
11. Gian Luigi Ferrari, Roberto Guanciale, and Daniele Strollo. Event based service coordination over dynamic and heterogeneous networks. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lect. Notes in Comput. Sci.*, pages 453–458. Springer-Verlag, 2006.
12. Gianluigi Ferrari, Roberto Guanciale, and Daniele Strollo. Jscl: A middleware for service coordination. In *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2006.
13. Gianluigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Coordination via types in an event-based framework. *27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems FORTE*, 4574:66–80, 2007.
14. Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 249–259. ACM Press, 1987.
15. David Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
16. Object Management Group. Business process modelling notation. Technical report. http://www.bpmn.org.
17. Roberto Guanciale. *PhD Thesis*. PhD thesis, Institute for Advanced Studies, IMT, Lucca, Forthcoming, 2008.
18. Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. A calculus for service oriented computing. In *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.
19. A. Hosking, S. Jagannathan, J. Vitek, and A. Welc. A semantic framework for designer transactions. In *Proceedings of ESOP 2004*, volume 249–263 of *Lect. Notes in Comput. Sci.*, pages 124–138, 2004.
20. IBM. Business Process Execution Language (BPEL). Technical report, 2005.
21. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.
22. Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Department, Indiana University, 2003.

23. M. Mazzara and R. Lucchi. A framework for generic error handling in business processes. In *Proceedings of WS-FM 2004*, volume 105 of *Elect. Notes in Th. Comput. Sci.*, pages 133–145, 2004.
24. Jayadev Misra. A programming model for the orchestration of web services. In *SEFM*, pages 2–11. IEEE Computer Society, 2004.
25. OMG. Business Process Modeling Language. http://www.bpmi.org, 2002.
26. Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.
27. Daniele Strollo. *PhD Thesis*. PhD thesis, Institute for Advanced Studies, IMT, Lucca, Forthcoming, 2008.
28. W3C. Web Services Choreography Description Language (v.1.0). Technical report.
29. Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias M. Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-based development of service-oriented systems. In *FORTE 2006*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2006.

```
1  public class TransactionalComponent {
2   public TransactionalComponent (
3    ComponentAddress addr,
4    AtomicTask A,
5    AtomicTask C,
6    ComponentAddress[] prev,
7    ComponentAddress[] next,
8    )
9    {
10    super (addr); initReactions(A, C); initFlows(prev, next);
11   }
12
13  public void initFlows (
14    ComponentAddress[] prev,
15    ComponentAddress[] next)
16   {
17    for (int i=0; i<prev.length; i++){ createFlow (FW, next[i]); }
18    for (int i=0; i<prev.length; i++){ createFlow (RB, prev[i]); }
19   }
20
21  public void initReactions (AtomicTask A, AtomicTask C)
22   {
23    addReaction (
24     new Reaction (
25      // adds a lambda reaction related
26      // to forward topic.
27      new SignalType(FW, null),
28      new HandlerTask(){
29       public Object handle (Signal signal){
30          // Retreive the signal session topic as SC lambda binder
31          Object session = ((SignalType) (signal.getType())).getSession();
32        try {
33         A.exec(signal);
34        } catch (AtomicActionException e){
35         // if A internally fails,
36         // a signal with type RB is sent back
37         signal.getType().setTopic(RB);
38         emit(signal); return;
39        }
40        // installs a reaction to handle rollback
41        // coming backward and related to the current
42        // context.
43        // The compensation in installed only if
44        // the main activity has been correctly done
45        addReaction (
46         new Reaction (
47          // adds a check reaction related
48          // to forward topic.
49          new SignalType(RB, session),
50          new HandlerTask(){
51            public Object handle (Signal signal){
52             // executes the compensation
53             C.exec(signal); emit(signal);
54            }
55          })
56        );
57       }
58     })
59    );
60   }
61  }
```

Code 1: Transactional Component