Checking correctness of transactional behaviors [⋆]

Vincenzo Ciancia[1], Gian Luigi Ferrari[1], Roberto Guanciale[2] and Daniele Strollo[1,2]

[1] Università degli Studi di Pisa, Dipartimento di Informatica
Largo B. Pontecorvo 3 I-56127, Pisa, Italy
{ciancia,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

Abstract. The Signal Calculus is an asynchronous process calculus featuring multicast communication. It relies on explicit modeling of the communication structure of the network (communication flows), and on handling sessions, even *multi-party*. The calculus is strongly motivated by the practical needs of *Service-Oriented Computing*, and there exists a Java implementation, called JSCL, with a graphical modeling framework. To the aim of adding to `SC` (and JSCL) a verification environment, in this work we introduce the abstract semantics of `SC`, based on bisimulation. We show an example exploiting bisimilarity to prove the correctness of an `SC` model with respects to a transactional isolation requirement.

Keywords. Service Oriented Architectures, Event Notification, Coordination, Observational Equivalence

1 Introduction

The Service Oriented Architecture (SOA) [1] main challenge consists in the definition of an architectural style where applications are built by composition of distributed functionalities, called services, that can be accessed in a uniform and platform independent manner, and communicate with each other by exchanging messages. The Web Service (WS) platform has become the universally accepted mechanism for implementing SOAs. The main contribution of this technology relies on the adoption of XML (eXtensible Markup Language) that has opened a new perspective for developers and service providers enabling language and platform independence (a.k.a. *interoperability*). The Web Service core specifications provide mechanisms for describing, publishing, retrieving and accessing services.

An open issue, in WS world, is the definition of a language for describing how these services interact and to check if the related implementations adhere to the specifications. In our previous works, we provided and implemented a middleware, Java Signal Core Layer (JSCL), paired with a formal specification of the

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

programming facilities that it offers. At the abstract level, the middleware takes the form of the *Signal Calculus* (SC) [10,12,8], an high level language inspired by the asynchronous π-calculus [15] enriched with the concepts of component locality and the needed primitives for dealing with Event Notification (EN) paradigm [18] (namely, multicast channels, that also give rise to multi-party sessions).

The adoption of EN yields to model services in terms of reactive entities that, autonomously, declare the set of events they are interested in and the behavior that they perform upon their occurrence. The main advantages of EN adoption rely on loosely coupling of services and on its flexibility. Specifically, EN features high level coordination mechanisms that allow programmers/designers to decouple components and rely entirely on event handling.

In this work we focus our attention on the verification of SC protocols. For this purpose, we introduce an abstract semantics of SC networks, based on the notion of *bisimulation*, which not only represents the behavior of sets of components interacting with each other, but also that of isolated subsystems. Behavioral semantics is important because it allows to distinguish isolated components that behave differently when "plugged" into a network. Our semantics is inspired by the π-calculus "direct HT bisimulation" [15]. Exploiting the notion of bisimulation, SC systems can be verified against abstracted versions of their design.

In this paper, we outline the main features of our approach by considering a simple, but illustrative case study, described in [24]. The case study is modeled by taking into account the transactional requirements given at specification level, proving that constraints on transactional isolation are maintained in the involved components. The verification of the scenario is done by checking that it is bisimilar to a "magic" property, i.e. an abstracted design that models properties of interest.

The paper is organized as follows. In Section 2 we review the main features and the operational semantics of the SC process calculus. Section 3 presents the abstract semantics of SCbased on a labeled transition system. Section 4 presents the case study, its abstract modeling and highlights how to exploit the bisimulation relation to prove transactional isolation of networks. Section 5 yields some concluding remarks.

2 Background: the Signal Calculus

In this section, we introduce the *signal calculus*. This is a process calculus suitable to describe service coordination, adopting the event notification paradigm. The communication mechanism is inspired by the asynchronous π-calculus. The calculus is centered around the notion of *component*, written as $a[B]^R_F$ and representing a service uniquely identified by a name a, the public address of the service, having internal behavior B, interfaces R, called *reactions*, and outgoing connections F, called *flows*.

We assume a countable set $\mathcal{T}$ of *topic* names (ranged over by τ), representing the available signal types, and a countable set of component names, ranged over by $a, b, c, \ldots$. The notation $\vec{a}$ indicates a set of component names.

Components exchange messages, called *signals*, in the form of pairs of topics $\tau©\tau'$, where the first part is the signal type (which is, an unique name identifying an event kind), and the second one is a session identifier. Session identifiers and event kinds are freely interchangeable, and can be either freshly generated or received as input by reactions. When an event is raised by a component, it is notified to the components interested in handling it. Components are thus modeled in terms of reactive agents which declare, and can dynamically alter, the kind of events they are capable to handle.

Reactions describe available methods of a service in a given state. Their syntax is given by the following grammar:

$$R \quad ::= \quad 0 \quad | \quad \langle\alpha\rangle \rightarrow B \quad | \quad R|R$$

The *input prefix* $\langle\alpha\rangle$ is either $\tau©\lambda\tau'$ or $\tau©\tau'$, where τ' is bound in $\tau©\lambda\tau'$. The *lambda reaction* $\tau©\lambda\tau' \rightarrow B$ is triggered by signals having topic τ independently from their session, and binds τ' to the received session identifier. Conversely, the *check reaction* $\tau©\tau' \rightarrow B$ reacts only to signals having topic τ issued for the specific session τ'. Once a signal reaction takes place, the behavior B will be executed in the component in parallel with the current internal behavior. *Reaction composition* $R|R$ allows a component to react to different kinds of signal in different ways. The *empty reaction* 0 cannot respond to any signal.

Each component has a *flow* describing the *choreography*, from the point of view of the component. Flows describe addressees of messages, for each topic τ. Flow syntax is defined as follows:

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

where the *empty flow* 0 does not deliver any kind of signal, the *single flow* $\tau \rightsquigarrow \vec{a}$ delivers signals having topic τ to the components specified in the set $\vec{a}$. Finally, new flows can be appended to component interfaces by using the parallel composition construct $F|F$.

Now, we introduce the syntax of *behaviors*, the basic programs that each service executes when a reaction is triggered by signals. Behaviors are described by the following grammar:

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

The $\mathbf{out}\langle\tau©\tau'\rangle.B$ primitive spawns a signal of topic τ having session τ', and then continues as B. A number of copies of the same message are created inside the network, one for each component listed in the flow of the component, for the topic τ. Topics can be freshly generated using *topic restriction*, a binder that declares local topics; 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 appends F to its flows. The empty and parallel constructs have the obvious meaning.

Networks describe the component distribution and carry signals exchanged among components. Network syntax is defined as follows:

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

A network can be empty $\emptyset$, a single component $a[B]_F^R$, the parallel composition of networks $N \parallel N'$, or the restriction of a topic in a (sub)network. Networks carry signals exchanged among components. The signal emission spawns into the network, for each target component, an "envelope" $\langle \tau \copyright \tau' \rangle @ a$ containing the signal and the target component name a. Finally, the last production allows to extend the scope of freshly generated topics over networks.

We assume that each service is identified by an unique name, and each name identifies at most one service, as it is usual in service-oriented computing.

We define a *network context* as a network having an "hole" where another network can be "plugged in". Formally, contexts are the terms generated by the grammar below, having only one occurrence of the symbol $-$:

$$C ::= \emptyset \quad | \quad a[B]_F^R \quad | \quad C \parallel C \quad | \quad \langle \tau \copyright \tau' \rangle @ a \quad | \quad (\nu\tau)C \mid -$$

The well formedness condition is also extended to contexts, so that a context is considered valid for a network when their component names are disjoint. This is formalized in the following definition.

Definition 1. *A network is* well formed *if the names of the components it contains are all different. We say that a context $C[-]$ is a* well formed context *of a network N if $C[N]$ is well formed.*

Free and bound names for networks, reactions, behaviors and flows are defined by structural induction in the usual way. We summarize the main rules in the following:

$$\begin{array}{ll}
fn(\tau \copyright \tau' \to B) = fn(B) \cup \{\tau, \tau'\} & bn(\tau \copyright \tau' \to B) = bn(B) \setminus \{\tau, \tau'\} \\
fn(\tau \copyright \lambda\tau' \to B) = fn(B) \setminus \{\tau'\} \cup \{\tau'\} & bn(\tau \copyright \lambda\tau' \to B) = bn(B) \cup \{\tau'\} \setminus \{\tau'\} \\
fn((\nu\tau)B) = fn(B) \setminus \{\tau\} & bn((\nu\tau)B) = bn(B) \cup \{\tau\} \\
fn((\nu\tau)N) = fn(B) \setminus \{\tau\} & bn((\nu\tau)N) = bn(B) \cup \{\tau\}
\end{array}$$

We define structural congruence over the syntax of the calculus as the smallest congruence that satisfies the commutative monoidal laws for $(R, |, 0)$, $(F, |, 0)$, $(B, \ | \ , 0)$ and $(N, \|, \emptyset)$, α-conversion of bound names, and the rule s below. In particular, notice that τ is not in the scope of τ' in $\tau \copyright \lambda\tau' \to B$.

$$\frac{N \to N'}{N \parallel M \to N' \parallel M}\ (npar)$$

$$\frac{a[B]_F^R \to a[B']_{F'}^{R'}}{a[B \mid B_1]_F^R \to a[B' \mid B_1]_{F'}^{R'}}\ (par) \qquad \frac{N \to N_1}{(\nu\tau)N \to (\nu\tau)N_1}\ (new)$$

$$a[\mathbf{rupd}\,(R')\,.B]_F^R \to a[B]_F^{R|R'} \quad (rupd) \qquad a[\mathbf{fupd}(F').B]_F^R \to a[B]_{F|F'}^R \quad (fupd)$$

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

$$\langle\tau \copyright \tau'\rangle @ a \parallel a[0]_F^{\tau \copyright \tau' \to B|R} \to a[B]_F^R \quad (check)$$

$$\langle\tau \copyright \tau'\rangle @ a \parallel a[0]_F^{\tau \copyright \lambda\tau_1 \to B'|R} \to a[\{\tau'/\tau_1\}B]_F^{\tau \copyright \lambda\tau_1 \to B'|R} \quad (lam)$$

Fig. 1. Operational semantics

$$(\nu\tau)0 \equiv 0 \qquad ((\nu\tau)B) \mid B' \equiv (\nu\tau)(B \mid B'),\ \text{if } \tau \notin fn(B')$$

$$(\nu\tau)(\nu\tau')B \equiv (\nu\tau')(\nu\tau)B \qquad (\nu\tau)(\nu\tau')N \equiv (\nu\tau')(\nu\tau)N$$

$$(\nu\tau)\emptyset \equiv a[0]_F^0 \equiv \emptyset \qquad ((\nu\tau)N) \parallel N' \equiv (\nu\tau)(N \parallel N'),\ \text{if } \tau \notin fn(N')$$

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

2.1 Reaction Rules

We briefly recall the reduction semantics of SC [12]. This is defined using the previously introduced structural congruence and the *flow projection* function $((F){\downarrow_\tau})$, defined as

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

This function takes a flow and a topic and yields the set of target component names to which signals having topic τ have to be delivered.

The reduction semantics of `SC` explains how components, at each step, communicate and update their interface. The reduction relation $\to$ is depicted in Figure 1. We assume the set of rules to be augmented with structural congruence, i.e., the following additional rule is used:

$$\frac{N \equiv N' \quad N' \to M' \quad M' \equiv M}{N \to M}\ (struct)$$

Rules labeled *rupd* and $fupd$ update, respectively, reactions and flows of a process. Rule *emit* introduces in the network a new envelope for the event kind τ targeted to each subscribed component $((F)\downarrow_\tau = \{b_1, \dots, b_n\})$. Rules labeled *check* and *lam* model activation of *check* reactions, that exactly match the session identifier, and of *lambda* reactions, that receive a session identifier as argument. Rules *npar*, *struct* and *new* are usual in process calculi, while *par* allows behaviors to be added in parallel into a component, preserving reactions. This rule allows us to define the semantics only on components whose internal behavior has no parallel operation, avoiding the need for separate rules. This happens because synchronization of two internal behaviors of the same component is not possible in our framework.

3 LTS Semantics

Here we present the *behavioral* semantics of networks, in terms of a labeled transition system that represents not only the behavior of sets of components that interact with each other, but also of isolated subsystems. Having an LTS semantics is important because it allows to distinguish isolated components that behaves differently when inserted into a network (e.g. a component with an installed reaction, and the empty component).

The transition system is similar in spirit to work on the asynchronous π-calculus by Honda and Tokoro [15], and Amadio, Castellani and Sangiorgi [2]. The set of observable actions α is specified as follows:

$$\alpha ::= \emptyset \;\;|\;\; \langle \tau \copyright \tau' \rangle @a \;\;|\;\; \langle \tau \copyright (\tau') \rangle @a \;\;|\;\; \tau \copyright \tau' @a \;\;|\;\; \tau \copyright \tau' @(a)$$

In our syntax, $\emptyset$ models unobservable actions. $\langle \tau \copyright \tau' \rangle @a$ is *free* (asynchronous) output with event kind τ, session type τ' and addressee a. $\langle \tau \copyright (\tau') \rangle @a$ is *bound* output, and $\tau \copyright \tau' @a$ is free input. $\tau \copyright \tau' @(a)$ represents the action of receiving a message and storing it in parallel with the current process. This action is observable in any system, thus including the empty network. This behavior is the essence of asynchronous communication, and is similar to the transition rule named in_0 in [2], which is used to define the so-called "directed HT bisimulation", derived, on its turn, from the rules given in [15]. All names in the actions are free, with the exception of τ' in *bound* output action. Finally we use $n(\alpha)$ to denote the set of names occurred in the action α.

The labeled transition relation over networks is defined by the rules depicted in Figure 2. We briefly comment on the semantics. The *async* rule allows any system to perform an input, simply storing the received message for subsequent usage. The *out* rule makes observable the output capability of a system with pending messages. Rules *struct*, *par*, *rupd*, $fupd$, *new* and *npar* are very similar to their counterparts in the unlabeled semantics. Rules *check* and *lam* model the capability of a system to consume messages present on the network, the former strictly matching on the session identifier, and the latter receiving sessions as input. In a similar fashion to the π-calculus, *ext* and *bsync* model sending

$$\frac{N \equiv N' \quad N' \xrightarrow{\alpha} M' \quad M' \equiv M}{N \xrightarrow{\alpha} M}\ (struct)$$

$$\frac{}{a[\mathbf{rupd}\ (R')\ .B]^R_F \xrightarrow{\emptyset} a[B]^{R|R'}_F}\ (rupd) \qquad \frac{}{a[\mathbf{fupd}(F').B]^R_F \xrightarrow{\emptyset} a[B]^R_{F|F'}}\ (fupd)$$

$$\frac{(F)\downarrow_\tau = \{b_1, \dots, b_n\}}{a[\mathbf{out}\langle\tau \copyright \tau'\rangle.B']^R_F \xrightarrow{\emptyset} a[B']^R_F \parallel \langle\tau \copyright \tau'\rangle@b_1 \parallel \dots \parallel \langle\tau \copyright \tau'\rangle@b_n}\ (emit)$$

$$\frac{}{\langle\tau \copyright \tau'\rangle@a \xrightarrow{\langle\tau \copyright \tau'\rangle@a} \emptyset}\ (out) \qquad \frac{}{N \xrightarrow{\tau \copyright \tau'@(a)} N \parallel \langle\tau \copyright \tau'\rangle@a}\ (async)$$

$$\frac{R' = \tau \copyright \tau' \to B}{a[0]^{R|R'}_F \xrightarrow{\tau \copyright \tau'@a} a[B]^R_F}\ (check) \qquad \frac{R' = \tau \copyright \lambda\tau' \to B}{a[0]^{R|R'}_F \xrightarrow{\tau \copyright \tau''@a} a[\{\tau''/\tau'\}B]^{R|R'}_F}\ (lam)$$

$$\frac{N \xrightarrow{\alpha} N_1 \quad \tau \notin n(\alpha)}{(\nu\tau)N \xrightarrow{\alpha} (\nu\tau)N_1}\ (new) \qquad \frac{N \xrightarrow{\langle\tau \copyright \tau'\rangle@a} N' \quad \tau \neq \tau'}{(\nu\tau')N \xrightarrow{\langle\tau \copyright (\tau')\rangle@a} N'}\ (ext)$$

$$\frac{N \xrightarrow{\langle\tau \copyright (\tau')\rangle@a} N' \quad M \xrightarrow{\tau \copyright \tau'@a} M' \quad \tau' \notin fn(M)}{N \parallel M \xrightarrow{\emptyset} (\nu\tau')N' \parallel M'}\ (bsync)$$

$$\frac{N \xrightarrow{\langle\tau \copyright \tau'\rangle@a} N' \quad M \xrightarrow{\tau \copyright \tau'@a} M'}{N \parallel M \xrightarrow{\emptyset} N' \parallel M'}\ (sync)$$

$$\frac{a[B]^R_F \xrightarrow{\alpha} a[B']^{R'}_{F'}}{a[B \mid B_1]^R_F \xrightarrow{\alpha} a[B' \mid B_1]^{R'}_{F'}}\ (par) \qquad \frac{N \xrightarrow{\alpha} N' \quad bn(\alpha) \cap fn(M) = \emptyset}{N \parallel M \xrightarrow{\alpha} N' \parallel M}\ (npar)$$

Fig. 2. Behavioral semantics

a restricted name as an output message, and receiving it as a fresh name. Finally, rule *sync* allows communication by linking input reactions and output capabilities of pending messages.

Rule labeled with (*async*), first given by Amadio, Castellani and Sangiorgi in [2], is the essence of asynchronous communication. This rule allows any process (even those that do not perform input) to store a message without consuming it, so that one cannot directly observe *when* input actions actually happen. In the definition of bisimulation below, only asynchronous input transitions (that is, transitions obtained from the *async* rule) are kept in account, while "normal" input is not considered. This allows two processes that only differ in the way they interleave input with other actions to be considered bisimilar.

Even though they are similar, the semantics of the asynchronous π-calculus and that of SC differ in some key aspects. Namely, SC features dynamic multicast channels due to the dynamic nature of flows. Hence, the addressee of a message is not statically known. This is the reason why our calculus features the output primitive, that using rule (*out*) spawns a certain number of messages in parallel, while in the asynchronous π-calculus there is no such construct.

The notion of *weak* transition system is defined in the standard way:

$$\begin{aligned} N &\overset{\emptyset}{\Longrightarrow} N' \quad \text{iff } N(\xrightarrow{\emptyset})^* N' \\ N &\overset{\alpha}{\Longrightarrow} N' \quad \text{iff } N \overset{\emptyset}{\Longrightarrow} . \xrightarrow{\alpha} . \overset{\emptyset}{\Longrightarrow} N' \text{ for all } \alpha \neq \emptyset \end{aligned}$$

The following theorem establishes a link between the reduction relation and the observational semantics.

Theorem 1. *$N \rightarrow N'$ if and only if $N \xrightarrow{\emptyset} N'$.*

Finally we provide the definition of SC-bisimulation ($\sim_{\mathtt{SC}}$). This relation allows to distinguish isolated subsystems (e.g. a component, or a partition of a network) that behave differently when inserted into a network, even though, in isolation, they cannot react.

Definition 2. *$\sim_{\mathtt{SC}}$ is the largest symmetric relation on SC-terms such that if $N \sim_{\mathtt{SC}} M$, $N \xrightarrow{\alpha} N'$, $\alpha \neq \tau \copyright \tau' @ a$, $bn(\alpha) \cap fn(M) = \emptyset$ implies that $M \xrightarrow{\alpha} M'$ and $N' \sim_{\mathtt{SC}} M'$.*

The notion of weak SC bisimulation ($\approx_{\mathtt{SC}}$) is obtained substituting in the above definition the transition relation with the weak one.

Bisimulation allows one to check for properties that have to be satisfied by the implementation of a system against its design expressed in a high-level language. Sometimes the implementation is slightly modified in order to verify a subset of the system requirements, e.g. by inserting the implementation in a suitable *controlled* context or environment, where it can be formally shown that, by construction, only properties of interest can lead to violation of the design. We show an example of this technique in section 4, as an application of the behavioral modeling framework we are developing.

Theorem 2. *If $N \sim_{\mathtt{SC}} N'$ then*

$$N \parallel \langle \tau_1 \copyright \tau_1' \rangle @ a_1 \ldots \parallel \langle \tau_k \copyright \tau_k' \rangle @ a_k \sim_{\mathtt{SC}} N' \parallel \langle \tau_1 \copyright \tau_1' \rangle @ a_1 \ldots \parallel \langle \tau_k \copyright \tau_k' \rangle @ a_k$$

Proof. (outline) Since the rule *async* can be applied to any network and envelope, the network N can perform a transition step labeled $\alpha = \tau \copyright \tau' @ (a)$, going to $N \parallel \langle \tau \copyright \tau' \rangle @ a$. The same rule can be applied to the network N', that goes to $N' \parallel \langle \tau \copyright \tau' \rangle @ a$. Since N and N' are bisimilar, when they perform the same transition α, they must go in bisimilar state: $N \parallel \langle \tau \copyright \tau' \rangle @ a \sim_{\mathtt{SC}} N' \parallel \langle \tau \copyright \tau' \rangle @ a$. This proves that two bisimilar network remain bisimilar if composed with the same envelope. This proof can be applied with any number of envelopes, proving the theorem. □

Theorem 3. *For any context C, and any two networks N and N', such that $N \sim_{\mathtt{SC}} N'$, with C a well formed context of both networks (see Definition 1), it holds that $C(N) \sim_{\mathtt{SC}} C(N')$.*

4 The Car Repair Scenario

In this section we adopt the SC calculus to model the service coordination issues of the SENSORIA car repair scenario [24], consisting of a car manufacturer service offering assistance support to their customers.

4.1 The Sensoria scenario

A car manufacturer offers an assistance service to the customer once his/her car breaks down. Once contacted, such 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. Several services are involved into the system and interact to reach a common goal. Their inter-dependencies are summarized as follows:

– before any service lookup is made, the credit card is charged 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 tow truck and garage appointment.

This scenario can be described through a business process language. We use the industry standard Business Process Modeling Language (BPMN [13]) to graphically describe the scenario and the inter-dependencies among services. The BPMN model of this scenario is presented in Figure 3. Notice that the model exploits the transactional and compensation facilities of BPMN and that the car rental service is a sub-transaction, since it does not affect other activities. We briefly recall the graphical notation adopted in BPMN. A double-lined boundary indicates that the sub-process is a transaction. The single-lined boxes represent activities executed inside transactions and the activities linked through backward arrows represent the related compensation activities that must be executed when the process is rolling back. The blank circles represent the entry and exit points of a transaction. Finally, the diamond containing the plus symbol represents the joining of two activities. The full BMPN specification can be found in [13].

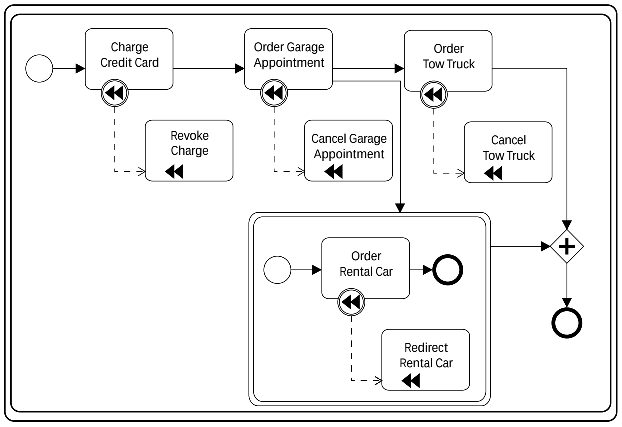

Fig. 3. Car repair scenario: the BPMN model

4.2 Modeling the Car Repair Scenario

Services involved into the Car Repair Scenario (CRS) scenario are described by `SC` components. To specify the interactions among participants, we introduce the following signal topics:

- τ_f is used to propagate *forward signals* to inform components about the completion of previous activities;
- τ_r is used to propagate *rollback signals* to components. Such signals are treated by executing the compensation activity and subsequently by propagating, backwards, the signal to the other participants;
- τ_n is used to implement the join mechanism among parallel activities executed inside the same workflow session.
- τ_{ok} is used internally by components to represent the successful termination of an activity.
- τ_{exc} is used internally by components to represent an internal failure, for example the throwing of an exception.

In `SC`, *transactional components* can be described as services reacting to both τ_f and τ_r notifications. At the reception of a τ_f signal, the component executes its main activity and installs the corresponding compensation reaction. At the reception of a τ_r signal, the previously installed compensation is executed. We suppose that each invocation of the transactional workflow has a unique session (in the following referred as τ). The consumer has to generate the session, that will be delivered with each signal to identify the workflow instance. Notice that, for a workflow session, the compensation activity must be executed only after the successful execution of the main activity. A transactional component having address a, a main activity A and a compensation C is translated to an `SC` model by the function TC. The connections to other components are described by the sets *next* and *prev* containing the target components to which, respectively, τ_f and τ_r signals must be forwarded. The TC function is defined as follows:

$$TC(a, A, C, prev, next) \triangleq (\nu\tau_{ok})(\nu\tau_{exc})a[0]^{R_{TC}(A,C)}_{F_{TC}(a,next,prev)}$$

where:

$$\begin{array}{ll}
F_{TC}(a, next, prev) & \triangleq \tau_f \rightsquigarrow next|\tau_r \rightsquigarrow prev|\tau_{exc} \rightsquigarrow a|\tau_{ok} \rightsquigarrow a \\
R_{TC}(A, C) & \triangleq \tau_f ©\lambda\tau \rightarrow \mathbf{rupd}\,(R_{res}(C)) \ | \ A \\
R_{res}(C) & \triangleq \tau_{ok} ©\tau \rightarrow B_{ok}(C)|\tau_{exc} ©\tau \rightarrow B_{exc} \\
B_{ok}(C) & \triangleq \mathbf{rupd}\,(R_{rb1}(C))\,.\mathbf{out}\langle\tau_f ©\tau\rangle.0 \\
R_{rb1}(C) & \triangleq \tau_r ©\tau \rightarrow C \\
B_{exc} & \triangleq \begin{cases} \mathbf{rupd}\,(R_{rb2})\,.\mathbf{out}\langle\tau_f ©\tau\rangle.0 & \text{subtransaction} \\ \mathbf{out}\langle\tau_r ©\tau\rangle.0 & \text{otherwise} \end{cases} \\
R_{rb2} & \triangleq \tau_r ©\tau \rightarrow \mathbf{out}\langle\tau_r ©\tau\rangle.0
\end{array}$$

Initially, the component has an installed reaction (R_{TC}) for handling the forward flow (τ_f notifications). Once the reaction is activated, it retrieves the signal session, that identifies the workflow instance, and executes the main activity A. The formalization of the activity A and of the compensation C are out of our scope; hereafter, we assume that:

1. if the main activity A successfully terminates, a signal τ_{ok} is internally raised, to inform the component that the flow can continue
2. if the main activity A fails, a signal τ_{exc} is internally raised, informing the component to start the backward flow
3. the last operation of the compensation C is the rising of a rollback signal ($\mathbf{out}\langle\tau_r ©\tau\rangle.0$).

Notice that the topics τ_{ok} and τ_{exc} are restricted to the local scope of component. Concurrently with the activity A, the component installs the reactions, defined by R_{res}, to check the termination state of A (τ_{ok} or τ_{exc}).

If the activity A successes, it internally delivers a τ_{ok} signal and the behavior B_{ok} is executed. It installs a check reaction (R_{rb1}), that is used to wait for a rollback notification from a successor component, and propagates the τ_f signal to the next components in the workflow (using $\mathbf{out}\langle\tau_f ©\tau\rangle.0$). If, later, a τ_r signal for the session τ is received, the compensation C is executed and the rollback signal is propagated to previous stages (since we suppose that the last operation of the compensation is $\mathbf{out}\langle\tau_r ©\tau\rangle.0$).

If the activity A fails, it internally delivers a τ_{exc} signal and the behavior B_{exc} is executed. Notice that two implementation of the behavior are provided: the first one is used if the component acts as an isolated sub-transaction (e.g. car rental service), while the second one is used if the components acts as a standard transactional activity. In the first case, the behavior propagates the τ_f signal, since an error of the sub-transaction should not affect the computation of the other components. Moreover the behavior installs a reaction for τ_r that just propagate the backward flow. In the second case, the behavior simply starts the backward flow, raising a rollback signal.

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

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

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

The emitter represents the entry point of the parallel branch. Essentially it activates the forward flow of *next* components, representing the parallel activities, and synchronizes their backward flows. The synchronization mechanism is implemented by sequentially installing two reactions for the topic τ_r and the session τ (through **rupd** ($\tau_r © \tau \rightarrow$ **rupd** ($\tau_r © \tau \rightarrow ...$))). After that the synchronization mechanism has been installed, the emitter activates the forward flow ($\mathbf{out}\langle\tau_n © \tau\rangle.\mathbf{out}\langle\tau_f © \tau\rangle.0$). 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 ($\mathbf{out}\langle\tau_r © \tau\rangle.0$) 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 needs to be notified about the session τ via a τ_n signal. This is necessary since there is not mutual exclusion among executed behaviors.

The car repair scenario can be modeled by the following SC network:

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

Notice that τ_r events raised by the *truck* component are notified to the *car* service, since an error occurred in the execution of a main activity must activate the compensations of all other concurrent components. Instead the τ_r events raised by the *car* component are notified only to the emitter, since car is a sub transaction.

4.3 Checking sub-transaction isolation

As discussed above, the rental car service is an isolated sub-transaction, namely, if the car rental fails, it should not affect the execution of the other components in the network. Regardless of the implementation details of the main activity and of the compensation, we model only the signal emissions that represent their termination. Hence, the car service that fails (Car_{exc}) and the other one that successes (Car_{ok}) are modeled as:

$$\begin{array}{ll} Car_{exc} & \triangleq TC(car, \mathbf{out}\langle \tau_{exc} \copyright \tau_s \rangle.0, \mathbf{out}\langle \tau_r \copyright \tau_s \rangle.0, \{e\}, \{c\}) \\ Car_{ok} & \triangleq TC(car, \mathbf{out}\langle \tau_{ok} \copyright \tau_s \rangle.0, \mathbf{out}\langle \tau_r \copyright \tau_s \rangle.0, \{e\}, \{c\}) \end{array}$$

Now we prove the *transaction isolation property* of the car service by comparing its model with a *magic* car service. This is a transactional component that performs the ideal behavior: when it receives a τ_f signal, it propagates the signal to *next* components, while, when it receives a τ_r signal, it propagates the signal to *prev* components. Then, we check that, independently from the behavior executed internally by the car service, the whole transactional workflow performs the same action of the one containing the *magic* service. Formally the workflow containing the Car_{exc} (or Car_{ok}) must be bisimilar to the one containing the *magic* car service. This service can be model as:

$$\begin{array}{ll} Car_{magic} & \triangleq car[0]^{\tau_f \copyright \lambda\tau \rightarrow skip.(\,\dots\,).skip.\mathbf{out}\langle \tau_f \copyright \tau \rangle.\mathbf{rupd}(\tau_r \copyright \tau \rightarrow \mathbf{out}\langle \tau_r \copyright \tau \rangle.0)}_{\tau_f \rightsquigarrow next | \tau_r \rightsquigarrow prev} \\ skip.B & \triangleq \mathbf{fupd}(0) \end{array}$$

In the above process, the *skip* action is used for internal computation steps. However, this is not a primitive of the calculus, but rather it is a derived operation, modeled by installation of an empty flow (hence, not altering the flow of the component).

The process describes a set of possible magic properties, parametrized by the number of *skip* actions. For the system to satisfy the required property, it is sufficient that there exists a number of *skip* actions that lets the bisimulation check succeed. We use the compositionality property of the bisimilarity (Theorem 3) as a "substitution principle": the statement $Car_{ok} \sim_{\mathtt{SC}} Car_{magic}$ (and $Car_{exc} \sim_{\mathtt{SC}} Car_{magic}$) ensures that the bisimulation result propagates to the whole workflows.

5 Future Work

We have presented an LTS semantics for the SC process calculus. The obtained abstract semantics, based on bisimulation, allows one to reason about behavioral properties of SC networks. The SC-JSCL framework has been designed 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 [16], BPML [22],WS-CDL [23]). The strict interplay between SC and JSCL permits to drive and verify the 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 [17], Global Calculus [5], λ_{req} [3] ORC [19], SCC [4], SOCK [14] to cite a few), since it focuses on a lower level of abstraction, merging the theoretical formalization with the implementation requirements. Indeed, the emphasis in SC-JSCL relies on the notion of event notification that strictly fits to the loosely coupling nature of services.

We foresee two development lines. In this work, bisimulation proofs have been done by hand, while one would expect automated checkers to be used. The fresh name generation construct of SC, even though giving it great expressive power (in particular, for the possibility to handle new sessions), makes it difficult to define and implement finite state algorithms for bisimulation checking and (in perspective) model checking. *History-Dependent* automata [20] are an operational model where garbage-collection of unused names can be exploited to obtain finite state models of systems featuring generation of fresh resources [7]. As a possible future development, thus, it would be interesting to express the semantics of SC using HD-automata, in order to be able to reuse work on minimization and bisimulation checking algorithms for nominal calculi [9].

In [12], we introduced an algebraic structure over topics. This allows us to implement more complex coordination logics directly encoded inside the signal type. The definition of bisimulation in this case should make use of the algebraic structure to obtain a suitable *quantitative* notion of bisimulation, allowing to express properties of a system with respects to e.g. a range of security policies. On the logical side, there is a close connection, which should be studied in detail, with the quantitative/spatial logic over c-semirings defined in [6].

The SC/JSCL framework is equipped with a programming environment, called *JSCL4Eclipse* [11], that allows one to graphically model JSCL networks and to automatically generate the stub implementation. As a long term research goal, we aim to integrate verification tools based on bisimulation and model checking techniques within our development framework.

References

1. Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors. *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings.* ACM, 2004.
2. Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998.
3. M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino. Secure service orchestration. In *FOSAD*, volume 4667 of *Lecture Notes in Computer Science.* Springer, 2007.
4. 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.

5. 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.
6. Vincenzo Ciancia and Gian Luigi Ferrari. Co-Algebraic Models for Quantitative Spatial Logics. In *Quantitative Aspects of Programming Languages (QAPL'07)*, 2007.
7. Vincenzo Ciancia and Ugo Montanari. A name abstraction functor for named sets. In *Coalgebraic Methods in Computer Science 2008*. to appear.
8. 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 *Lecture Notes in Computer Science*, pages 453–458. Springer-Verlag, 2006.
9. Gian Luigi Ferrari, Ugo Montanari, and Emilio Tuosto. Coalgebraic minimization of hd-automata for the pi-calculus using polymorphic types. *Theor. Comput. Sci.*, 331(2-3):325–365, 2005.
10. Gianluigi Ferrari, Roberto Guanciale, and Daniele Strollo. Jscl: A middleware for service coordination. In Najm et al. [21], pages 46–60.
11. Gianluigi Ferrari, Roberto Guanciale, and Daniele Strollo. An Eclipse plugin for designing and developing Web Service orchestrations in JSCL. Technical report, 2007.
12. GianLuigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Coordination via types in an event-based framework. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.
13. Object Management Group. Business process modelling notation. Technical report. http://www.bpmn.org.
14. 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.
15. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.
16. IBM. Business Process Execution Language (BPEL). Technical report, 2005.
17. 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.
18. Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report 574, Department of Computer Science, Indiana University.
19. Jayadev Misra. A programming model for the orchestration of web services. In *SEFM*, pages 2–11. IEEE Computer Society, 2004.
20. Ugo Montanari and Marco Pistore. History Dependent Automata. Technical report, Dipartimento di Informatica, Università di Pisa, 1998. TR-11-98.
21. Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors. *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, volume 4229 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
22. OMG. Business Process Modeling Language. http://www.bpmi.org, 2002.
23. W3C. Web Services Choreography Description Language (v.1.0). Technical report.
24. Martin Wirsing, Allan Clark, Stephen Gilmore, Matthias M. Hölzl, Alexander Knapp, Nora Koch, and Andreas Schroeder. Semantic-based development of service-oriented systems. In Najm et al. [21], pages 24–45.