Coordination via Types in an Event-based Framework*

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

[1] Dipartimento di Informatica,
Università degli Studi di Pisa, Italy
{giangi,strollo}@di.unipi.it
[2] Istituto Alti Studi IMT Lucca, Italy
{roberto.guanciale,daniele.strollo}@imtlucca.it
[3] Computer Science Department, University of Leicester
et52@mcs.le.ac.uk

Abstract. We propose a novel approach to service choreography through a typed process calculus that features an event notification paradigm for coordinating distributed components (e.g., services). Basically, the type system expresses coordination policies for handling the events spawn in a network so that distributed components react to events when the type of their public interface is "compatible" with (the policies expressed by) the types of signals.
Remarkably, the type system can naturally handle *multi-party sessions*, as shown in the formalisation of the OpenID protocol which requires multi-party sessions for handling user identities

1 Introduction

A well known paradigm for programming/modeling distributed systems is *event notification* (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. An important characteristic that discriminates EN systems lays in how the middleware dispatches events. Two main approaches are possible: *topic-based* and *content-based* mechanisms [5,16].

The dispatching mechanism in topic-based (also known as *subject-based*) EN systems is simpler than in content-based systems. In topic-based EN systems, events are categorized into topics which subscribers register to. When an event belonging to a topic τ is emitted, all the components subscribed for τ will eventually react to the event. Notice that publishers and subscribers have to know the topics at hand. In content-based EN, component decoupling is enforced by allowing subscribers to register for events satisfying a given *property*. When an event is emitted the middleware has to dispatch it to *all* the subscribers whose property holds on that event (an example of content-based is SIENA [4]). Notoriously, content-based dispatching mechanisms must be efficient because notification sets, i.e. the set of subscribers that must be notified for the event, can be order of magnitude larger than in topic-based EN [6,18]. A main advantage of

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA and by the Italian FIRB Project TOCAI.IT.

content-based EN is that publishers and subscribers do not have to share any *a priori* knowledge about the topics. Subscribers use, instead, a language for expressing properties on events that publishers must simply accomplish with when emitting their events.A more abstract content-based model is the so called *type-based* EN [9] where topics are replaced by types (in a suitable type language). Typed events are also used in commercial middlewares (see [9] and the references therein).

This paper considers the Signal Calculus (SC) [11], a topic-based EN process calculus, and recasts it into a type-based framework, the eXtended Signal Calculus (XSC). The XSC calculus is a "typed version" of SC where events are emitted with types that coordinate publishers/subscribers interactions. For instance, an XSC publisher can emit an event with type $\tau \times \tau'$ that should be received by subscribers that can react to events of type τ *and* τ'. XSC types have a twofold role. First, typing allows subscribers to filter their events of interest (as usual in type-based EN). Second, publishers exploit type information to specify which (kind of) subscribers should react to events. For instance, in the previous example, a subscriber that is able to react only to events of type τ will not be capable of reacting to an event $\tau \times \tau'$. The way types are used is indeed the main original contribution of XSC with respect to standard type-based EN systems.

A further advantage of XSC is that types allow us to handle *sessions* so that a sort of "virtual communication link" among publishers and subscribers can be established despite they do not need to know each other's names. Intuitively, a session identifies the scope within which an event is significant: partners that are not in this scope cannot react to events of the session. Furthermore, the session handling mechanisms provided by XSC can deal with multi-party sessions in a natural way. At the best of our knowledge, multi-party sessions are ruled out from other approaches. For instance, in [13,3,2] only two-party sessions are tackled. Indeed, these proposals aim to model the basic use of sessions as done in many protocols of e.g. the IP-stack (TCP, HTTP, etc.). We argue that XSC complements these approaches by providing higher-level constructs on sessions that allow a closer formalization of more abstract protocols where multi-party sessions are relevant.

To demonstrate the adequacy of our approach, we apply XSC to specify the OpenID protocol [17], a complex protocol for managing distributed identities whose behavior requires many parties to participate to the same session. XSC mechanisms have allowed us to identify and formally specify *all the assumptions* underlying the definition of the OpenID protocol. We argue that our approach will make easier to reason and verify properties of protocols requiring multi-party sessions.

The main effort of this paper is on the formal definition of XSC showing its adequacy to handle complex coordination policies via typing information. This is part of an ongoing work on the design, implementation and experimental evaluation of a middleware, called JSCL [11], supporting coordination policies for service-oriented applications. The distinguished feature of our approach resides in the close interplay between formal definition and implementation: the implementation of the JSCL middleware is driven by the formal definition of the (X)SC calculus.

Structure of the paper Section 2 reviews the basic features of the SC calculus. Section 3 introduces the concept of multi-party sessions on events and shows how it yields a synchronization mechanism. Section 4 introduces XSC types. The operational semantics

of XSC is presented in Section 5. In Section 6 we specify the OpenID protocol. Finally, Section 7 gives some concluding remarks.

2 Preliminaries: Signal Calculus

The *Signal Calculus* (SC) is a process calculus introduced in [11] as a foundational model of the JSCL (after Java Signal Core Layer) programming middleware, for coordinating distributed components (e.g., web services). SC relies on the EN paradigm where *components*, the basic building blocks of SC, interact by issuing/reacting to *events*. A component represents a 'simple' service interacting via asynchronous signal passing. Each component is identified by a unique name, which, intuitively, can be thought of as the URI of the published service. The signals exchanged among components are messages containing information regarding the managed resources and the events raised during internal computations. Signals are classified by *topics*; specifically, each component specifies (i) the reaction to activate on reception of signals of a certain topic and (ii) the set of event flows, namely the collection of component names the emitted signals will be delivered. Hence, while reactions define the interacting behavior of the component, flows define the component view of the coordination policies. The SC primitives allow one to *dynamically* modify the topology of the coordination policies by adding new flows and reactions to components.

Standard EN paradigms rely on brokered communication; SC, instead, adopts a non-brokered notification mechanism where subscription and emission are *explicitly* tagged with naming information, e.g. the name of the target components. This avoids any centralization point by distributing the connection managing to each involved participant. Brokered EN paradigms are more appropriate when coordination is handled by an orchestrator, while non-brokered approaches fit much better when choreography is adopted. For a detailed comparison among brokered and non-brokered EN see [14].

The adoption of the EN paradigm, for managing coordination policies has two main advantages. On the one hand, it is a well known programming model and, on the other hand, it permits the distribution of coordination activities and of the underlying computational infrastructure. This distribution is obtained by decoupling publishers and subscribers. The intuitive idea is that publishers and subscribers do not rely on any 'a priori' knowledge.

The dynamic flavor of the SC calculus permits modeling a wide range of coordination policies for service-oriented applications (e.g. in [10] the primitives have been used to deal with dynamic and heterogeneous networks). However, other primitives providing high-level abstractions for programming are desirable. In particular, in the current formulation, information associated to signals is not structured and topics cannot be created dynamically. Furthermore, the notion of session abstraction is missing: components cannot keep track of concurrent event notifications.

3 Extended Signal Calculus

In this section, we present an extension of the SC calculus, called XSC, that permits managing of sessions and is also capable of handling structured topics via suitable types.

3.1 Managing Sessions

The calculus is centered around the notion of *component*. A component $a[B]_F^R$ is a service identified by a unique name a: the public address of the service. The expression B describes service internal behavior. Expressions R and F, called *reactions* and *flows*, respectively, have to be thought of as the service interface. We assume a set of *topic* names Λ (ranged over by τ), a set of signal variables (ranged over by x) and a set of signal names (ranged over by $s, s_1, s_2 \ldots$). Signal names represent data exchanged among components and should carry additional information even if this feature is not explicitly modeled. Finally, we assume a set of component names $a, b, \ldots$. Hereafter, we adopt the notation $\vec{a}$ to denote a set of component names.

The syntax of behaviors is given by the following grammar,

$$
\begin{array}{lll}
B ::= 0 \mid B \mid B' \mid !B \mid & & \\
\quad \mid \ \overline{s} : \tau \copyright \tau'.B' & & \textit{(Signal emission)} \\
\quad \mid \ \nu\tau.B' & & \textit{(Topic creation)} \\
\quad \mid \ +\lceil x : \tau \copyright \lambda\tau' \to B \rceil .B' & & \textit{(Lambda reaction)} \\
\quad \mid \ +\lceil x : \tau \copyright \tau' \to B \rceil .B' & & \textit{(Check reaction)} \\
\quad \mid \ +\lfloor \tau \rightsquigarrow \vec{a} \rfloor .B' & & \textit{(Flow update)}
\end{array}
$$

where the productions in the first row have the usual process algebraic meaning. A *signal emission* $\overline{s} : \tau \copyright \tau'.B'$ describes the emission of the signal s of topic τ over the session identified by the topic τ'. Topics can be freshly generated using the *topic creation* primitive. A *lambda reaction* $+\lceil x : \tau \copyright \lambda\tau' \to B \rceil .B'$ installs a "generic reaction" for the topic τ in the component interface; this reaction handles all signals with topic τ, regardless of their session. In the reaction behavior B, τ' and x are bound by the lambda reaction[4]. After the installation of a reaction the continuation B' is executed. Conversely, *check reaction* installs a reaction that can handle only signals having the topic τ issued for the session τ' and, in this case, only x is bound in the reaction behavior B. A *flow update* $+\lfloor \tau \rightsquigarrow \vec{a} \rfloor .B$ extends the flow of a component, specifying the set of component names $\vec{a}$ to which deliver signals having topic τ. After the installation of a flow, the behavior B' is executed.

Reactions and flows syntax have the following syntax:

$$
\begin{array}{lll}
R ::= \ 0 \mid R|R & & \\
\quad \mid \ x : \tau \copyright \lambda\tau' \to B & & \textit{(Lambda reaction)} \\
\quad \mid \ x : \tau \copyright \tau' \to B & & \textit{(Check reaction)}
\end{array}
\qquad
\begin{array}{lll}
F ::= 0 \mid F|F & & \\
\quad \mid \ \tau \rightsquigarrow \vec{a} & & \textit{(single flow)}
\end{array}
$$

[4] See Appendix A for a formal definition of free and bound names of the binders of XSC.

where the *empty reaction (resp. flow)* 0 cannot respond to any signal (resp. cannot emit a signal for any receiver) and *reaction (resp. flow) composition* $R|R$ allows a component to react to (resp. to emit) different kinds of signal. Reactions R_1 and R_2 are called *subreactions* of the reaction composition $R_1|R_2$.

Reactions describe how a component reacts upon the reception of a signal. As pointed out before, a lambda reaction is triggered by signals independently from their session, while a check reaction reacts only to signals in the session τ'. Once a reaction to a signal takes place, the behavior B will be executed in the component in parallel with the existing behaviors. Flows describe the component view of the choreography: a component with a single flow $\tau \rightsquigarrow \vec{a}$ can deliver signals of topic τ to components specified in $\vec{a}$.

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

$$N ::= \mathbb{0} \mid a[B]_F^R \mid N \| N \mid \langle s : t \copyright \tau @ a \rangle \mid \nu\tau.N$$

A network can be empty $\mathbb{0}$, a single component $a[B]_F^R$, or the parallel composition of networks $N \| N'$. Networks carry signals exchanged among components. The signal emission spawns into the network, for each target component, an "envelope" $\langle s : t \copyright \tau @ a \rangle$ containing the signal and the name a of the target component. Finally, the last production allows to extend the scope of freshly generated topics over networks.

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

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

and, whenever $B \equiv B'$:

$$+\lceil x : \tau \copyright \lambda\tau' \to B \rceil.B'' \equiv +\lceil x : \tau \copyright \lambda\tau' \to B' \rceil.B''$$
$$+\lceil x : \tau \copyright \tau' \to B \rceil.B'' \equiv +\lceil x : \tau \copyright \tau' \to B' \rceil.B''$$

If $B \equiv B'$, the following rules hold for structural congruence over reactions:

$$x : \tau \copyright \lambda\tau' \to B \equiv x : \tau \copyright \lambda\tau' \to B'$$
$$x : \tau \copyright \tau' \to B \equiv x : \tau \copyright \tau' \to B'$$

Similarly, $\equiv$ is the smallest equivalence relation that respects the commutative monoidal laws for $(N, \|, \mathbb{0})$ and the following ones:

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

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

To give an intuition of the features and the facilities of XSC, we consider a simple scenario. The operational semantics will be presented in Section 4.

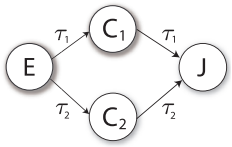

Fig. 1. An example of synchronization between two components

3.2 Joining events

Since XSC components are autonomous entities communicating through asynchronous primitives, it could be useful to introduce a lightweight synchronization mechanism that allows us to express that a task can be executed whenever other concurrent tasks have been completed. In this scenario we show how to encode a form of join synchronization among concurrent tasks.

Figure 1 shows an emitter E, two intermediate components C_1 and C_2, and the join service J. The emitter E starts the communications raising two events of different topics toward C_1 and C_2 that perform an internal computation and then notify their termination by issuing an event to the join service. The component J waits that both the intermediate services have completed their tasks and then executes its internal behavior B. The signals sent to C_1 and C_2 are both related to the same session τ that is later used by J to apply the synchronization on the same workflow. Clearly, 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 execution.

This example can be modeled by the XSC network $E\|C_1\|C_2\|J$, where:

$$
\begin{aligned}
E &\triangleq e[\nu\tau.\bar{s} : \tau_1 ⓒ \tau.\bar{s} : \tau_2 ⓒ \tau.0]^0_{\tau_1 \rightsquigarrow c_1 | \tau_2 \rightsquigarrow c_2} \\
C_i &\triangleq c_i[0]^{x:\tau_i ⓒ \lambda\tau \to \bar{x}:\tau_i ⓒ \tau.0}_{\tau_i \rightsquigarrow j}, \qquad\qquad i = 1,2 \\
J &\triangleq j[0]^{x:\tau_1 ⓒ \lambda\tau \to + \lceil x':\tau_2 ⓒ \tau \to B \rceil .0}_0
\end{aligned}
$$

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. When such kind of signal is received, the proper behavior B is executed. Notice that the creation of the specialized reaction for the τ_2 implies that a possible pendent envelope is consumed.

4 Structured Topics

We have described how the session mechanism permits to specify complex coordination policies by constraining the ways components may react to notification of events.

$$
\begin{array}{ll}
t' \times t'' \equiv t'' \times t' & t' + t'' \equiv t'' + t' \\
t \times t \equiv t & t + t \equiv t \\
t' \times (t'' \times t''') \equiv (t' \times t'') \times t''' & t' + (t'' + t''') \equiv (t' + t'') + t''' \\
t \times \star \equiv t & t + \varepsilon \equiv t \\
t \times \varepsilon \equiv \varepsilon & t + \star \equiv \star \\
t \times (t' + t'') \equiv (t \times t') + (t \times t'') &
\end{array}
$$

Fig. 2. Structural congruence over topics

The basic idea is to control the coordination workflow by exploiting information about topics and sessions to trigger the execution of the suitable reactions. In this section, we further develop this idea by introducing some operators on topics that induce an algebraic structure on events. We then show how the algebraic structure on events can be used to have a finer control over the coordination activities of components.

We define the signal topic t as follows:

$$t ::= \varepsilon \mid \star \mid \tau \mid t \times t \mid t + t$$

The constant topics ε and $\star$ are used to define the *empty* and the *global* event kinds, respectively. Intuitively, a signal having an empty topic can be consumed by a reaction having an empty behavior. A signal having a global topic can be handled by any component, activating any reaction. Signal topics can be composed using the constructors $\times$ and $+$. A signal having topic $t \times t'$ can be consumed only by components that can handle both event kinds t and t'. Moreover a signal having topic $t + t'$ can be consumed by any component that can handle event kinds t or t'. The constructors $+$ and $\times$ can be informally interpreted as logical *disjunction* and *conjunction*.

The formal definition of the meaning of structured topics is given algebraically by introducing a structural congruence over them (see Figure 2). Notice that the $\times$ and $+$ are associative, commutative and idempotent. Also, $\times$ distributes over $+$, moreover, $\star$ and ε are their respective neutral elements. For instance, $t \times \star \equiv t$ and $t + \varepsilon \equiv t$ states that a signal of topic $t \times \star$ or $t + \varepsilon$ activates the same reactions activated by signals having topic t; similarly $t \times \varepsilon \equiv \varepsilon$ states that a signal of topic $t \times \varepsilon$ cannot activate any reaction, while $t + \star \equiv \star$ states that a signal of topic $t + \star$ activates any reaction. Formally, the algebraic structure over topic takes the form of a C-Semiring [1].

Preorder relation. *The binary relation $\sqsubseteq$ over topics is the least preorder satisfying the following axioms:*

$$t \sqsubseteq \varepsilon, \qquad \star \sqsubseteq t, \qquad t \sqsubseteq t, \qquad t \sqsubseteq t \times t', \qquad t + t' \sqsubseteq t$$

Intuitively the preorder $t_1 \sqsubseteq t_2$ formalizes the idea that the topic t_1 is less restrictive than the topic t_2. For example, a signal having topic $\tau_1 + \tau_2$ triggers either a reaction for τ_1 or one for τ_2. Hence, the coordination policy expressed by $\tau_1 + \tau_2$ is less restrictive than the one expressed by τ_1.

The algebraic structure over topics allows us to define the policies to aggregate events. The XSC syntax of behaviors can be extended to deal with the structure of topics by simply refining the signal emission primitive as $\overline{s} : t \textcircled{c} \tau'.B'$, where t represents the

signal topic. We have now to specify the way a component may react upon the reception of a signal of a certain topic. In other words, a main question, here, is to understand which reactions a component may dynamically activate to match the policy specified by the topics of events. In this paper, we will answer this question by introducing a suitable type system over component reactions. The type system allows us to precisely identify the set of reactions matching a given event topic.

Conversation types (ranged over by T) classify signals by their topic structures (policies) and sessions. Their syntax is defined below:

$$\begin{array}{rll} T ::= & t\textcircled{c}\tau \mid & \textit{(Session conversation type)} \\ & t\textcircled{c}\star & \textit{(Generic conversation type)} \end{array}$$

A *session conversation type* $t\textcircled{c}\tau$ characterizes signals (of a topic t) *within* a session τ. A *generic conversation type* $t\textcircled{c}\star$ captures the notion of signals (of a topic t) not belonging to a specific session.

Conversation types are equivalent if the structures of their topics and their sessions are equivalent. Formally, equations in Figure 2 are extended with the following rules:

$$\frac{t \equiv t'}{t\textcircled{c}\tau \equiv t'\textcircled{c}\tau} \qquad \frac{t \equiv t'}{t\textcircled{c}\star \equiv t'\textcircled{c}\star}$$

Conversation types can be equipped with a subtype relation which will be used to formalize how signals are consumed by reactions. Namely, if $T \sqsubseteq T'$ then reactions able to consume signals with conversation type T' can consume signals with conversation type T as well.

Subtype relation. *The subtype relation $T \sqsubseteq T'$ over conversation types is defined as the smallest preorder relation that satisfies the following inference rules:*

$$\frac{t \sqsubseteq t'}{t\textcircled{c}\tau \sqsubseteq t'\textcircled{c}\tau}\ (1) \qquad \frac{t \sqsubseteq t'}{t\textcircled{c}\tau \sqsubseteq t'\textcircled{c}\star}\ (2) \qquad \frac{t \sqsubseteq t'}{t\textcircled{c}\star \sqsubseteq t'\textcircled{c}\star}\ (3)$$

Rules (1) and (3) have a clear interpretation in terms of the preorder over topics. Rule (2) is controvariant wrt the session part of the conversation type and formalizes the idea that a lambda reaction can be activated by signals independently by their session.

A *reaction type* is a (possibly empty) set of conversation types and describes the set of signals that can be consumed by a reaction.

Reaction typing. A reaction R has reaction type $\mathbb{T}$ *when* $\vdash R : \mathbb{T}$ *can be inferred from the following rules:*

$$\frac{}{\vdash 0 : \emptyset}\ (1) \qquad \frac{}{\vdash x : \tau\textcircled{c}\tau' \to B : \{\tau\textcircled{c}\tau'\}}\ (2)$$

$$\frac{}{\vdash x : \tau\textcircled{c}\lambda\tau' \to B : \{\tau\textcircled{c}\star\}}\ (3) \qquad \frac{\vdash R_1 : \mathbb{T}_1 \qquad \vdash R_2 : \mathbb{T}_2}{\vdash R_1 | R_2 : \mathbb{T}_1 \cup \mathbb{T}_2}\ (4)$$

Rules $(1 \div 4)$ are quite natural; for instance, rule (3) states that the type of a lambda

reaction $x:\tau©\lambda\tau' \to B$ is the singleton $\{\tau©\star\}$. Reaction types have a natural subtype relation given by the subset inclusion ($\mathbb{T} \subseteq \mathbb{T}'$).

Given a non-empty reaction type $\mathbb{T} = \{\tau_1©r_1,\ldots,\tau_n©r_n : r_i \in \Lambda \cup \{\star\} \text{ for } i = 1,\ldots,n\}$, we let

$$^{\times}\mathbb{T} = \tau_1 \times \ldots \times \tau_n, \quad \mathbb{T}^{\times} = r_1 \times \ldots \times r_n, \qquad ^{+}\mathbb{T} = \tau_1 + \ldots + \tau_n, \quad \mathbb{T}^{+} = r_1 + \ldots + r_n$$

while $^{\times}\mathbb{T} = \star = \mathbb{T}^{\times}$ and $^{+}\mathbb{T} = \varepsilon = \mathbb{T}^{+}$ if $\mathbb{T} = \emptyset$. The following properties trivially hold.

$$^{\times}\mathbb{T} = \star \Leftrightarrow \mathbb{T} = \emptyset \qquad \mathbb{T}^{\times} = \tau \Rightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \forall r_i.r_i \in \{\tau,\star\})$$
$$\mathbb{T}^{+} = \varepsilon \Leftrightarrow \mathbb{T} = \emptyset \qquad \mathbb{T}^{\times} = \star \Leftrightarrow (\mathbb{T} = \emptyset \ \vee\ \forall r_i.r_i = \star)$$
$$\mathbb{T}^{+} = \star \Leftrightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \exists r_i.r_i = \star) \qquad \mathbb{T}^{+} = \tau \Leftrightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \forall r_i.r_i = \tau)$$

After having defined the preorder on topics and the subtype relation for conversation types, we define a formal mechanism that establishes when a reaction is *enabled* to handle a signal reception. This definition is the basic tool that will be exploited at runtime to activate the reaction matching an event notification.

Reaction enabling. *Let $T \equiv t©\tau$ be a conversation type and $\mathbb{T}$ a non empty reaction type. We say that* reactions with type $\mathbb{T}$ can be activated by signals with conversation type *T, and we write $T \Subset \mathbb{T}$, if the following conditions hold:*

1. $t \sqsubseteq {}^{\times}\mathbb{T}$ *and* $\mathbb{T}^{\times} \sqsubseteq \tau$
2. $\forall \mathbb{T}' \subset \mathbb{T}.\mathbb{T}' \not\equiv \emptyset \implies \mathbb{T}'$ *does not enjoy the Condition 1*

Condition 1 expresses that the topic of the signals is less restrictive than the conjunction of the topics of the reactions ($t \sqsubseteq^{\times} \mathbb{T}$) and, since $\mathbb{T}$ is not empty then it is of the form $\{\tau_1©r_1,\ldots,\tau_n©r_n : r_i \in \Lambda \cup \{\star\} \text{ for } i = 1,\ldots,n\}$, reactions waiting for a session topic different from τ cannot be activated because $\forall i.r_i \equiv \tau \ \vee\ r_i \equiv \star$. Condition 2 ensures that enabled reactions are *minimal*, namely, that each subreaction ($\forall \mathbb{T}' \subset \mathbb{T}$) cannot be activated by signals having signal type T. The following table gives examples where conditions 1 and 2 hold or not.

Conversation Type $t©\tau$	Reaction Type $\mathbb{T}$	$t \sqsubseteq {}^{\times}\mathbb{T}$	$\mathbb{T}^{\times} \sqsubseteq \tau$	Cond. 2
$\tau_1+\tau_2©\tau$	$\{\tau_1©\tau\}$	$\surd$	$\surd$	$\surd$
$\tau_1\times\tau_2©\tau$	$\{\tau_1©\tau,\tau_2©\star\}$	$\surd$	$\surd$	$\surd$
$\tau_1\times\tau_2©\tau$	$\{\tau_1©\tau\}$	$\times$	$\surd$	$\surd$
$\tau_1©\tau$	$\{\tau_1©\tau'\}$	$\surd$	$\times$	$\surd$
$\tau_1+\tau_2©\tau$	$\{\tau_1©\tau,\tau_2©\star\}$	$\surd$	$\surd$	$\times$
$\tau_1\times\tau_2©\tau$	$\{\tau_1©\tau,\tau_2©\star,\tau_3©\star\}$	$\surd$	$\surd$	$\times$

Enabled reaction set. *Given a reaction R, the* set of enabled subreactions by a conversation type *$t©\tau$ is defined as* $R_{t©\tau} = \{R'.R \equiv R'|R'' \wedge \vdash R' : \mathbb{T} \ \wedge\ t©\tau \Subset \mathbb{T}\}$.
Let R_1 be $x:\tau_1©\tau \to B_1$ and R_2 be $x:\tau_2©\lambda\tau' \to B_2$, examples exploiting the enabled reaction set are given in Table 1. Notice that in the second row of Table 1 only one reaction $(R_1|R_2)$ is enabled. Upon reception of a signal having conversation type T, both

Conversation Type $t©\tau$	Reaction R	$R_{t©\tau}$
$\tau_1+\tau_2©\tau$	R_1	$\{R_1\}$
$\tau_1\times\tau_2©\tau$	$R_1\|R_2$	$\{R_1\|R_2\}$
$\tau_1\times\tau_2©\tau$	R_1	$\emptyset$
$\tau_1+\tau_2©\tau$	$R_1\|R_2$	$\{R_1,R_2\}$

Table 1. Enabled reaction set example

subreactions R_1 and R_2 will be concurrently activated. Also, in the fourth row of Table 1 two different reactions (R_1 and R_2) are enabled. Upon the reception of a signal having conversation type T, only one of them will be activated nondeterministically.

Preferred reactions. *Let R be a reaction and $t©\tau$ be a session conversation type. The set of* preferred reactions in R wrt $t©\tau$ *is defined as:*

$$R_{t©\tau\downarrow} = \left\{ R_1 \in R_{t©\tau}. \vdash R_1 : \mathbb{T}_1 \Rightarrow \forall R_2 \in R_{t©\tau}. \vdash R_2 : \mathbb{T}_2 \Rightarrow \begin{pmatrix} \mathbb{T}_1^+ \equiv \tau \\ \vee \\ \mathbb{T}_2^\times \sqsubseteq \mathbb{T}_1^\times \end{pmatrix} \right\}$$

Basically, each reaction $R_1 \in R_{t©\tau\downarrow}$ is composed only by check reactions for the τ session or, if it is composed only by lambda reactions, then it cannot exists another subreaction composed by check reactions for τ.

The topic structures can be adopted to model the example described in Section 3.2, refining the emitter component as $E \triangleq e[\nu\tau.\bar{s} : \tau_1+\tau_2©\tau.0]^0_{\tau_1\rightsquigarrow c_1|\tau_2\rightsquigarrow c_2}$.

5 Operational Semantics

The operational semantics of XSC is given in the classical reduction style and exploits the structural congruences defined in Section 3.1. Some auxiliary functions on flows and reactions are introduced for simplifying the definition of the reduction relation on networks.

The *flow projection*, $(F)\downarrow_t$, defined as

$$\begin{array}{ll} (\tau\rightsquigarrow\vec{a})\downarrow_\tau = \vec{a} & (\tau\rightsquigarrow\vec{a})\downarrow_{\tau'} = (\tau\rightsquigarrow a)\downarrow_\varepsilon = (0)\downarrow_t = \emptyset \\ (\tau\rightsquigarrow\vec{a})\downarrow_\star = \vec{a} & (F_1|F_2)\downarrow_t = (F_1)\downarrow_t \cup (F_2)\downarrow_t \\ (F)\downarrow_{t_1+t_2} = (F)\downarrow_{t_1} \cup (F)\downarrow_{t_2} & (F)\downarrow_{t_1\times t_2} = (F)\downarrow_{t_1} \cap (F)\downarrow_{t_2} \end{array}$$

takes a flow and a topic and yields the set of target component names for the topic t.

The *reaction projection*, $(R)\downarrow_{s:T}$, defined as

$$\begin{array}{l} (0)\downarrow_{s:\star} = (0,0) \\ (x:\tau'©\tau''\to B)\downarrow_{s:t©\tau} = (\{s/x\}B,0) \\ (x:\tau'©\lambda\tau''\to B)\downarrow_{s:t©\tau} = (\{s/x,\tau/\tau''\}B, x:\tau'©\lambda\tau''\to B) \\ (R_1|R_2)\downarrow_{s:t©\tau} = (B' \mid B'', R'|R''), \text{ if } (R_1)\downarrow_{s:t©\tau} = (B',R') \text{ and } (R_2)\downarrow_{s:t©\tau} = (B'',R'') \end{array}$$

takes a reaction R and a signal s typed by T and returns a pair (B,R') such that B is the behavior of R instantiated with s and R' is the reaction to be installed. Notice

$$\frac{}{a[+\lceil x:\tau \copyright \lambda\tau' \to B\rceil.B' \mid B'']_F^R \to a[B' \mid B'']_F^{R|x:\tau\copyright\lambda\tau'\to B}}\ (\text{RLambaUpd})$$

$$\frac{}{a[+\lceil x:\tau \copyright \tau' \to B\rceil.B' \mid B'']_F^R \to a[B' \mid B'']_F^{R|x:\tau\copyright\tau'\to B}}\ (\text{RCheckUpd})$$

$$\frac{}{a[+\lfloor \tau \rightsquigarrow b\rfloor.B \mid B']_F^R \to a[B \mid B']_{F|\tau\rightsquigarrow b}^R}\ (\text{FlowUpd})$$

$$\frac{(F)\downarrow_{t\copyright\tau} = \vec{b}}{a[\bar{s}:t\copyright\tau.B]_F^R \to a[B]_F^R \| \Sigma_{c_i\in\vec{b}}\langle s:t\copyright\tau@c_i\rangle}\ (\text{Emit})$$

$$\frac{R \equiv R'|R_0 \quad R' \in R_{t\copyright\tau\downarrow} \quad (R')\downarrow_{s:t\copyright\tau} = (B', R'')}{\langle s:t\copyright\tau@a\rangle \| a[B]_F^R \to a[B|B']_F^{R_0|R''}}\ (\text{RActivation}) \qquad \frac{N \to N'}{N\|N_1 \to N'\|N_1}\ (\text{NStep})$$

Fig. 3. Operational semantics

that reaction projection permits to consume check reactions and to maintain lambda reactions installed. Also, reaction projection is applied, by construction, to reactions that can consume the signal s. This assumption is guaranteed by the reduction rules using the type system.

The reduction relation $\to$ over networks is defined in Figure 3. Reactions can be added to a component by executing the behavioral primitives RLambaUpd and RCheckUpd. These primitives change the interface of a by appending to the set of installed reactions the new one. The only difference between the two primitives regards the kind of reaction installed. Analogously the FlowUpd updates the flow interface of a component by appending new target component names. The Emit and RActivation rules define notification dispatching: at emission time, component a spawns into the network a signal targeted to all the components ($c_i \in \vec{b}$) subscribed for the signal type (according to the $(F)\downarrow_t$ projection). Once a signal envelop has been spawn into the network the RActivation rule can be applied to the target component; the application of this rule activates, non deterministically, a reaction among the ones in the reaction projection $R' \in R_{t\copyright\tau\downarrow}$. Then, the activated reaction is replaced in the interface of a by R'' reaction obtained by applying the reaction projection.

6 Federated Identity Example

In order to illustrate the main facilities made available by the XSC calculus, in this section we show an example involving multi-party sessions. A typical scenario in which several agents are involved into the same session is represented by user-centric digital identity systems. We consider an application of the OpenID protocol, an open framework for distributed identity management. The solution presented can be easy adapted for similar systems e.g., i-Name [15] and Microsoft CardSpace [7].

The main advantage of the identity management systems is the unique identification of the user agent on the network in the same manner an URI uniquely identifies a

website. To reach this goal, these systems define a special kind of services, called *identity providers*, that act as intermediate agents among service consumers and providers. Another key feature offered by OpenID is the decentralization of the authentication protocol decoupling the service from a particular identity provider.

Hereafter, we denote a service consumer as C, an identity provider as IP and a service provider as SP. The protocol consists of two phases. In the first phase, C accesses its IP to be authenticated and to establish a private session. In the second phase C accesses a service SP specifying its identity and the IP that certifies her/his credentials. Notice that the actual authentication mechanism is not part of the specification of OpenID, and so it will not be treated: here we only deal with the message exchanges among the involved parties.

We start by giving the informal description of the OpenID protocol:

1. C initiates authentication with IP by presenting its credentials.
2. IP verifies user credentials and generates a new session shared with C. The session will be used to identify C.
3. C initiates authentication by presenting a User-Supplied Identifier to the SP via its User-Agent.
4. SP establishes an Endpoint URL used by C for authentication.
5. SP redirects the User-Agent of C to IP with an authentication request.
6. IP establishes whether C is authorized to perform authentication and wishes to do so. The way C authenticates to IP and any authentication policy are out of scope for OpenID.
7. IP redirects the User-Agent of C back to SP with either an assertion stating that the authentication is approved or a message that the authentication failed.
8. SP verifies the information received from the IP.

The OpenID protocol can be formally specified as the XSC network $C \| IP \| SP$ where C, IP and SP are the components defined in Figure 4. Notice that we omit to model the data exchanged among components, because we focus on the session exchanges and message sequences.

The user sends its credentials to the Identity Provider, rising an *Auth* event (via the B_c behavior). Notice that the client creates a new reaction to receive an event corresponding to the successful authentication (*AuthOK*) from the identity provider.

When the identity provider receives an authentication request (*Auth* event), it generates a new session (s_{ip}). This will be used later to identify the user agent without an explicit communication of the user credentials. The service provider raises a successful authentication event (*AuthOK*), communicating the generated session. Notice that we assume that the user authentication is always successful, therefore we do not model the implementation verification of the user credentials. Finally, the identity provider creates a new reaction to receive a delegation event. This reaction can be activated only for the generated session. Only the authenticated user owning this session can generate a signal that can be consumed by this reaction.

When the user has been notified about the successful authentication, by receiving the session shared with the identity provider ($B_{AuthOK}(s_{ip})$), it can access to a federated service. The user communicates the claimed identity (*identifier*) (and not the whole credentials) to the service provider rising a *Claim* event.

$$
\begin{array}{ll}
C & \triangleq c[B_c]^0_{Auth\rightsquigarrow i|Claim\rightsquigarrow s|Delegate\rightsquigarrow i} \\
B_c & \triangleq \nu r.+\lceil x : AuthOK©\lambda s_{ip} \rightarrow B_{AuthOK}(s_{ip})\rceil \\
 & \quad .\overline{credentials} : Auth©r.0 \\
B_{AuthOK}(s_{ip}) & \triangleq \nu r.+\lceil x : Redirect_{sp}©\lambda s_{sp} \rightarrow B_{Redirect_{si}}(s_{ip}, s_{sp})\rceil \\
 & \quad .\overline{identifier} : Claim©r.0 \\
B_{Redirect_{si}}(s_{ip}, s_{sp}) & \triangleq +\lceil x : Redirect_{si}©\lambda s_3 \rightarrow B_{Redirect_{is}}(s_{ip}, s_{sp}, s_3)\rceil . \\
 & \quad \overline{x} : Delegate©s_{ip}.0 \\
B_{Redirect_{is}}(s_{ip}, s_{sp}, s_3) & \triangleq +\lfloor s_{sp} \rightsquigarrow s\rfloor .\overline{p} : s_{sp}©s_3.0 \\
 & \\
IP & \triangleq i[0]^{R_{ip}}_{AuthOK\rightsquigarrow c|Redirect_{ip}\rightsquigarrow c|Verified\rightsquigarrow s} \\
R_{ip} & \triangleq x : Auth©\lambda r \rightarrow B_{Auth}(r) \\
B_{Auth}(r) & \triangleq \nu s_{ip}.+\lceil x : Delegate©s_{ip} \rightarrow B_{Delegate}(s_{ip})\rceil . \\
 & \quad \overline{x} : AuthOK©s_{ip}.0 \\
B_{Delegate}(s_{ip}) & \triangleq \nu s_3.+\lceil x : Verify©s_3 \rightarrow B_{Verify}\rceil .\overline{x} : Redirect_{ip}©s_3.0 \\
B_{Verify} & \triangleq \overline{x} : Verified©s_3.0 \\
 & \\
SP & \triangleq s[0]^{R_{sp}}_{Redirect_{si}\rightsquigarrow c|Verify\rightsquigarrow i} \\
R_{sp} & \triangleq x : Claim©\lambda r \rightarrow B_{Claim}(r) \\
B_{Claim}(r) & \triangleq \nu s_{sp}.+\lceil x : s_{sp}©\lambda s_3 \rightarrow B_{Check}(s_3)\rceil .\overline{x} : Redirect_{si}©s_{sp}.0 \\
B_{Check}(s_3) & \triangleq +\lceil x : Verified©s_3 \rightarrow B_{Verified}\rceil .\overline{x} : Verify©s_3.0
\end{array}
$$

Fig. 4. XSC specification of the OpenID protocol

When a service provider receives a *Claim* event, it delegates the authentication of the identity to the identity provider. This is performed redirecting the client to the identity provider. Observe that the service provider generates a new session s_{sp} that is communicated via the redirect request ($\overline{x} : Redirect_{si}©s_{sp}.0$). The generated session is used as a new event, the service provider waits this event to perform the authentication. In OpenID this is implemented through the generation of a user-specific URL.

When the user receives the *Claim* response and the session shared with the service provider s_{sp}, it forwards the request to the identity provider, delegating the authentication to it.

Finally, on reception of a delegate event for the authenticated user ($Delegate©s_{ip}$), the identity provider generates a three-party session s_3 and requests the user to forward it to the service provider. The identity provider and the service provider use this session to verify the user claim. If the verification is successful, the service provider continues according to $B_{Verified}$ after the reception of the consumer parameter p, i.e., the behavior representing the service supplied by *SP* which depends on the provided service and therefore it is not specified.

7 Concluding Remarks

We introduced a process calculus to handle multi-party sessions and coordination policies in an event-notification (EN) framework. Our approach is based on type informa-

tion that naturally support and extend typed-based EN systems. We demonstrated the adequacy of the approach by specifying the OpenID protocol.

As future work we plan to investigate which properties the XSC type system enjoys. We are also studying different interpretation for the algebraic structure of topics. For instance, by relaxing the idempotency of $\_ \times \_$ we get a theory which allows one to *count* the number of topics, thus leading to a notion of linear types. Finally, the type system described in this paper yields a *constraint semiring* structure [1] that has been successfully exploited to model QoS aspects of distributed systems [8,12]. We argue that this will allow us to express QoS driven coordination policy within our type system.

We also plan to validate and assess our approach on a variety of languages for programming service coordination policies. A step toward this goal would be to encode the Global Calculus [3] in XSC.

At the implementation level, the JSCL middleware (see Section 2) has already been extended with some of the new concepts of XSC (e.g., logical ports and signal sessions), while topic creation and structured topic composition are under development.

References

1. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
2. 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 Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer-Verlag, 2006.
3. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2007.
4. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Annual Symposium on Principles of Distributed Computing PODC*, pages 219–227, 2000.
5. Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, volume 2538 of *Lecture Notes in Computer Science*, pages 59–68, London, UK, 2002. Springer-Verlag.
6. Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors, *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, pages 163–174. ACM Press, 2003.
7. David Chappell. Introducing windows cardspace. MSDN Library. Available at http://msdn2.microsoft.com/en-us/library/aa480189.aspx.
8. Rocco De Nicola, Gianluigi Ferrari, Ugo Montanari, Rosario Pugliese, and Emilio Tuosto. A Basic Calculus for Modelling Service Level Agreements. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *Coordination*, volume 3454 of *Lecture Notes in Computer Science*, pages 33 – 48. Springer-Verlag, April 2005.

9. Patrick Th. Eugster and Rachid Guerraoui. Distributed programming with typed events. *IEEE Software*, 21(2):56–64, March/April 2004.
10. Gianluigi 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.
11. Gianluigi Ferrari, Roberto Guanciale, and Daniele Strollo. Jscl: A middleware for service coordination. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2006.
12. Dan Hirsch and Emilio Tuosto. SHReQ: A Framework for Coordinating Application Level QoS. In K. Aichernig Bernhard and Beckert Bernhard, editors, *3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 425–434. IEEE Computer Society, 2005.
13. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. *Lecture Notes in Computer Science*, 1381:122–141, 1998.
14. Yi Huang and Dennis Gannon. A comparative study of web services-based event notification specifications. In *ICPP Workshops*, pages 7–14. IEEE Computer Society, 2006.
15. i-name specifications. Available at http://www.inames.net/developers.html.
16. Ying Liu and Beth Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Department, Indiana University, 2003.
17. David Recordon and Brad Fitzpatrick. *OpenID Authentication 1.1*. Available at http://openid.net/specs/openid-authentication-1_1.html.
18. David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In Karl Aberer, Vana Kalogeraki, and Manolis Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2003.

A Free names

We define the free names of our syntactic categories in the usual way:

$$
\begin{array}{ll}
fn(0) & = \emptyset \\
fn(!B) & = fn(B) \\
fn(+\lfloor \tau \rightsquigarrow a \rfloor.B') & = \{\tau, a\} \cup fn(B') \\
fn(B_1 \mid B_2) & = fn(B_1) \cup fn(B_2) \\
fn(+\lceil x : \tau \copyright \tau' \to B \rceil.B') & = fn(B) \setminus \{x\} \cup \{\tau, \tau'\} \cup fn(B') \\
fn(+\lceil x : \tau \copyright \lambda\tau' \to B \rceil.B') & = fn(B) \setminus \{x, \tau'\} \cup \{\tau\} \cup fn(B') \\
fn(\overline{s} : t \copyright \tau.B') & = fn(B') \cup \{s, \tau\} \cup fn(t) \\
fn(\nu\tau.B') & = fn(B') \setminus \{\tau\}
\end{array}
$$

$$
\begin{array}{ll}
fn(0) & = \emptyset \\
fn(R_1 | R_2) & = fn(R_1) \cup fn(R_2) \\
fn(x : \tau \copyright \tau' \to B) & = fn(B) \setminus \{x\} \cup \{\tau, \tau'\} \\
fn(x : \tau \copyright \lambda\tau' \to B) & = fn(B) \setminus \{x, \tau'\} \cup \{\tau\} \\
fn(0) & = \emptyset \\
fn(F_1 | F_2) & = fn(F_1) \cup fn(F_2) \\
fn(\tau \rightsquigarrow b) & = \{\tau, b\}
\end{array}
$$

$$
\begin{array}{ll}
fn(\emptyset) & = \emptyset \\
fn(\nu\tau.N) & = fn(N) \setminus \{\tau\} \\
fn(\langle s : t \copyright \tau @ a \rangle) & = \{s, a, \tau\} \cup fn(t) \\
fn(N_1 \| N_2) & = fn(N_1) \cup fn(N_2) \\
fn(a[B]_F^R) & = fn(B) \cup fn(F) \cup fn(R) \cup \{a\}
\end{array}
$$

$$
\begin{array}{ll}
fn(\tau) & = \{\tau\} \\
fn(\varepsilon) = fn(\star) & = \emptyset \\
fn(t_1 \copyright \tau) & = fn(t_1) \cup \{\tau\} \\
fn(t_1 \copyright \star) & = fn(t_1) \\
fn(t_1 \times t_2) = fn(t_1 + t_2) & = fn(t_1) \cup fn(t_2)
\end{array}
$$