

Coordination via Types in an Event-based Framework

Gianluigi Ferrari¹, Roberto Guanciale², Daniele Strollo^{1,2}, and Emilio Tuosto³

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

Abstract. We propose a novel approach to service choreography, formalized as a process calculus, called eXtended Signal Calculus (XSC). The XSC calculus features an event/notification paradigm for coordinating distributed components (e.g., services) through typed events. 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 in) the types of signals.

Also, the type system allows us to handle *sessions*. Remarkably, the proposed session handling mechanisms of XSC can deal with multi-party sessions in a natural way.

We give a formal semantics of XSC and show how it can be used to model the OpenID protocol, designed for handling user identities. Multi-party sessions are paramount importance in the OpenID protocol.

1 Introduction

A quite 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 request 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 [10, 19].

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.) Content-based EN systems enforce components decoupling

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 [9]). Notoriously, content-based dispatching mechanisms must be efficient because notification sets⁴ can be order of magnitude larger than in topic-based EN [11, 21]. An advantage of content-based EN with respect to topic-based is that publishers and subscribers do not have to share any *a priori* knowledge about the topics. Rather, subscribers use a language for expressing properties on events that the publisher must simply accomplish with when emitting their events. A more abstract content-based model is the so called *type-based* EN [13] where topics are replaced by types (in a suitable type language). Typed events are also used in commercial middlewares (see [13] for an account). Mostly, typed events are used to let programmers to specify properties using the public interface of events described by their type.

This paper considers the Signal Calculus (SC) [15], 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 signals of type τ and τ' (the $_ \times _$ constructor of our type system resembles intersection types and topics are basic types). The originality of XSC with respect to classic type-based EN is in how types are used. Indeed, types can be not only used by subscribers to filter their events of interest (as usual in type-based EN), but they can also be exploited by publishers to specify which (kind of) subscribers should react to events. In fact, in the previous example, a subscriber that is able to react only to events of type e.g., τ , will not be capable of reacting to the event $\tau \times \tau'$.

A further advantage of our approach is that types allow us to handle *sessions* that can be used to establish a sort of “direct communication” among publisher and subscribers. Intuitively, a session identify the scope within which an event is significant: partners that are not in this scope cannot react to events of the session. Session handling mechanisms of 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 approached dealing with sessions. For instance, in [17, 20, 7] only two-party sessions are tackled. In fact, all these proposals aim to model the basic use of sessions as done in many different protocols of e.g. the IP-stack (TCP, HTTP, etc.). We argue that XSC complements these other approaches by providing more high-level constructs on sessions that allow a closer formalization of more abstract protocols where multi-party sessions are relevant. Indeed, we show how XSC can naturally model OpenID [3], a protocol for managing distributed identities whose specification requires that many parties participate to the same session.

⁴ A notification set for an event is the set of subscribers that must be notified for the event.

Remarkably, XSC provides mechanisms that allow us to model OpenID more faithfully while proposals like in [20, 7] can be thought of being lower-level implementation languages. This makes easier to reason and verify properties of protocols requiring multi-party sessions.

The paper is organized as follows: in Section 2 we recall the main concepts of SC. The Section 3 is so structured: in Section 3.1 we introduce the concept of multiparty sessions on signals and then, in Section 3.2, we exemplify the concept showing how this mechanism can be adopted to program synchronization. In Section 4 we introduce dynamic types to deal with notification mechanism whose operational semantics is presented in Section 5. Finally, in Section 6, we apply the multiparty session mechanism to model the OpenID protocol.

2 Preliminaries: Signal Calculus

The *Signal Calculus* (SC) is a process calculus that has been introduced in [15] as a foundational model of a programming middleware, called Java Signal Core Layer (JSCL), for coordinating distributed components (e.g., web services). SC relies on the *event notification* paradigm where components interact by issuing/reacting to *events*. The basic building blocks of the calculus are called components. A component represents a 'simple' service interacting via asynchronous signal passing. Each component is identified by a unique name, which, intuitively, can be thought 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 tagged with a meta type representing the class of events they belong to. Such meta type information is often referred to with the term *topic*. Each component declares the reaction that can be activated at the reception of a signal of a certain topic (i.e. once a signal of a well defined topic is received, the associated reaction is activated). Moreover, each component declares 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 style of event notification is strictly related to the specific coordination pattern chosen. For example, the standard Web services conversational styles, orchestration and choreography, can be obtained by constraining the way components are involved into the coordination steps.

The SC primitive constructs allow one to *dynamically* modify the topology of the coordination policies by adding new flows as well as new reactions for topics that previously were not handled.

The adoption of the event notification paradigm, for managing coordination policies has two main advantages. At a methodological level is a well known programming model and, moreover, it allows 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' acknowledgment on their

$B ::=$	$\bar{s} : \tau \textcircled{c} \tau'. B'$		<i>(Signal emission)</i>
	$\nu \tau. B'$		<i>(Topic creation)</i>
	$+ [x : \tau \textcircled{c} \lambda \tau' \rightarrow B]. B'$		<i>(Lambda reaction)</i>
	$+ [x : \tau \textcircled{c} \tau' \rightarrow B]. B'$		<i>(Check reaction)</i>
	$+ [\tau \rightsquigarrow \vec{a}]. B'$		<i>(Flow update)</i>
	$B \mid B'$		<i>(Parallel)</i>
	$!B$		<i>(Bang)</i>
	0		<i>(Empty behavior)</i>

Fig. 1. Behaviors

existence and availability. Finally, the standard event notification mechanism relies on anonymous signals (brokered communication), here, instead, we assume that subscription and emission are *explicitly* tagged with naming information, e.g. the name of the target components. For a detailed comparison among event based approaches see [18].

The dynamic flavor of the SC calculus permits modeling a wide range of coordination policies for service-oriented applications (e.g. in [14] 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: component cannot keep track of multiple concurrent event notifications.

3 Extended Signal Calculus

In this section, we present an extension of the SC calculus, called XSC, that allows 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 $s, s_1, s_2 \dots$). 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, \dots . Hereafter, we adopt the notation \vec{a} to denote a set of component names.

Figure 1 gives the syntax of behaviors. A *signal emission* $\bar{s} : \tau \textcircled{c} \tau'. B'$ describes the emission of the signal s of topic τ over the session identified by the

$R ::= 0$		$(Empty\ reaction)$
$x : \tau \odot \lambda \tau' \rightarrow B$		$(Lambda\ reaction)$
$x : \tau \odot \tau' \rightarrow B$		$(Check\ reaction)$
$R R$		$(Reaction\ composition)$

Fig. 2. Reactions

topic τ' . Topics can be freshly generated using the *topic creation* primitive. A *lambda reaction* $+ [x : \tau \odot \lambda \tau' \rightarrow B].B'$ installs a *generic reaction* for the topic τ in the component interface; this reaction handles all signals having topic τ , regardless of their session. In the continuation B' , $\lambda \tau'$ and x are bound by the lambda reaction. 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 continuation B' . Similarly, a *flow update* $+ [\tau \rightsquigarrow \vec{a}].B'$ 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. The empty, bang and parallel constructs have the obvious meaning.

Reactions describe how a component reacts upon the reception of a signal and their syntax is given in Figure 2. The *empty reaction* cannot respond to any 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. *Reaction composition* allows a component to react to different kinds of signal.

Flows describe the component view of the choreography. Flow syntax is defined as follows:

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

A component with an *empty flow* (0) does not deliver any kind of signals. A component with a *single flow* ($\tau \rightsquigarrow \vec{a}$) delivers signals having topic τ to the components specified in the set \vec{a} . A component can *compose* several flows to deliver different kinds of signal to different sets of components.

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

$$N ::= \emptyset \mid a[B]_F^R \mid N\|N \mid \langle s : t \odot \tau @ a \rangle \mid \nu \tau. N$$

A network can be empty \emptyset , 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” $\langle s : t \odot \tau @ a \rangle$ containing the signal and the target component name a . Finally, the last production allows to extend the scope of freshly generated topics over networks⁵.

⁵ The syntactic categories of our calculus contain several binders. The notion of free names is defined as usual and reported in Appendix A.

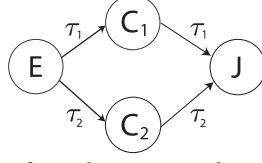


Fig. 3. An example of synchronization between two components

We define a structural congruence over reactions, flows and behaviors as the smallest congruence relation that satisfies the commutative monoidal laws for $(R, |, 0)$, $(F, |, 0)$ and $(B, |, 0)$.

Additionally, for the structural congruence over behaviors, the following laws hold:

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

and, whenever $B \equiv B'$:

$$\begin{aligned} +[x : \tau \odot \lambda \tau' \rightarrow B].B'' &\equiv +[x : \tau \odot \lambda \tau' \rightarrow B'].B'' \\ +[x : \tau \odot \tau' \rightarrow B].B'' &\equiv +[x : \tau \odot \tau' \rightarrow B'].B'' \end{aligned}$$

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

$$\begin{aligned} x : \tau \odot \lambda \tau' \rightarrow B &\equiv x : \tau \odot \lambda \tau' \rightarrow B' \\ x : \tau \odot \tau' \rightarrow B &\equiv x : \tau \odot \tau' \rightarrow B' \end{aligned}$$

Similarly, \equiv is the smallest equivalence relation that respects the commutative monoidal laws for $(N, \parallel, \emptyset)$ and the following ones:

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

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

Moreover, given a reaction R such that $R \equiv R_1 | R_2$, we refer to R_1 and R_2 as *subreactions* of 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.

3.2 Join Scenario

Since XSC components are autonomous entities communicating through asynchronous primitives, it could be useful to introduce a synchronization mechanism that allows 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 3 shows an emitter E , two intermediate components C_1 and C_2 , and the join component J . The emitter E starts the communications raising two different events toward the intermediate components. Both C_1 and C_2 perform

an internal computation and then forward the events to the join service, notifying their completion. 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:

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

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 event notification. 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 built over this idea by introducing an algebraic structure over topics thus inducing 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

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

Fig. 4. Structural congruence over topics

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 4). 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 $t + \varepsilon \equiv t$ state 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 $t + \star \equiv \star$ state that a signal of topic $t + \star$ activates any reaction. Formally, the algebraic structure over topic takes the form of a C-Semiring [5].

The XSC syntax of behaviors can be extended to deal with the structure for topics. We only need to refine the signal emission primitive as $\bar{s} : t \odot \tau'. B'$, where t represents the signal topic.

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

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

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. 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 event topic. Here, 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.

We first introduce the notion of conversation type (T) to classify signals. Syntax of conversation types is defined below:

$$\begin{array}{ll}
T ::= t \odot \tau' & \text{(Session conversation type)} \\
& t \odot \star & \text{(Generic conversation type)}
\end{array}$$

Conversation types classify signals by their topic structures (policies) and their sessions. A *session conversation type* $t\textcircled{\tau}'$ characterizes signals (of a topic t) within a session τ' . A *generic conversation type* $t\textcircled{\star}$ captures the notion of signals (of a topic t) not belonging to a specific session.

We say that two conversation types are equivalent if the structures of their topics and their sessions are equivalent. Formally, we extend the equations in Figure 4 with the following rules:

$$\frac{t \equiv t'}{t\textcircled{\tau} \equiv t'\textcircled{\tau}} \quad \frac{t \equiv t'}{t\textcircled{\star} \equiv t'\textcircled{\star}}$$

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:*

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

Rules (1) and (3) have a clear interpretation in terms of the preorder over topics. Rule (2) is contravariant with respect to the session part of the conversation type. The rule 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. *We say that a reaction R has reaction type \mathbb{T} , and we write $\vdash R : \mathbb{T}$, when it can be derived by applying the following inference rules:*

$$\frac{}{\vdash 0 : \emptyset} \quad (1) \quad \frac{}{\vdash x : \tau\textcircled{\tau}' \rightarrow B : \{\tau\textcircled{\tau}'\}} \quad (2)$$

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

Rules (1 ÷ 4) are quite natural; for instance, rule (3) states that the type of a lambda reaction $x : \tau\textcircled{\lambda}\tau' \rightarrow B$ is the singleton $\{\tau\textcircled{\star}\}$. Reaction types have a natural subtype relation given by the subset inclusion ($\mathbb{T} \subseteq \mathbb{T}'$).

We introduce some auxiliary notations for reaction types. Let \mathbb{T} be a non-empty reaction type $\mathbb{T} = \{\tau_1\textcircled{\tau}r_1, \dots, \tau_n\textcircled{\tau}r_n : r_i \in A \cup \{\star\} \text{ for } i = 1, \dots, n\}$, then

$$\begin{aligned} \times \mathbb{T} &= \tau_1 \times \dots \times \tau_n & \mathbb{T}^\times &= r_1 \times \dots \times r_n \\ + \mathbb{T} &= \tau_1 + \dots + \tau_n & \mathbb{T}^+ &= r_1 + \dots + r_n \end{aligned}$$

while we let $\times\mathbb{T} = \star = \mathbb{T}^\times$ and $+\mathbb{T} = \varepsilon = \mathbb{T}^+$ when $\mathbb{T} = \emptyset$. It is straightforward to prove the following properties:

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

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 run-time to activate the reaction matching an event notification.

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

1. $t \sqsubseteq \times\mathbb{T}$
2. $\mathbb{T}^\times \sqsubseteq \tau$
3. $\forall \mathbb{T}' \subset \mathbb{T} \wedge \mathbb{T}' \neq \emptyset$ and \mathbb{T}' does not enjoy the previous conditions.

We can now comment on conditions (1), (2) and (3) above.

- The condition (1) expresses that the topic of the signals (t) is less restrictive than the conjunction of the topics of the reactions ($\times\mathbb{T}$).
- Since \mathbb{T} is not empty then $\mathbb{T} = \{\tau_1\odot r_1, \dots, \tau_n\odot r_n : r_i \in A \cup \{\star\} \text{ for } i = 1, \dots, n\}$. The condition (2) is satisfied only if $\forall i.r_i \equiv \tau \vee r_i \equiv \star$. Hence, reactions waiting for a session topic different from τ cannot be activated.
- The condition (3) ensures that each subreaction ($\forall \mathbb{T}' \subset \mathbb{T}$) cannot be activated by signals having signal type T . This ensures that enabled reactions are *minimal*.

Examples exploiting the notion of *reaction enabling* are given in Table 1.

Coordination Type	Reaction Type	(1)	(2)	(3)
$\tau_1 + \tau_2\odot\tau$	$\{\tau_1\odot\tau\}$	✓	✓	✓
$\tau_1 \times \tau_2\odot\tau$	$\{\tau_1\odot\tau, \tau_2\odot\star\}$	✓	✓	✓
$\tau_1 \times \tau_2\odot\tau$	$\{\tau_1\odot\tau\}$	×	✓	✓
$\tau_1\odot\tau$	$\{\tau_1\odot\tau'\}$	✓	×	✓
$\tau_1 + \tau_2\odot\tau$	$\{\tau_1\odot\tau, \tau_2\odot\star\}$	✓	✓	×
$\tau_1 \times \tau_2\odot\tau$	$\{\tau_1\odot\tau, \tau_2\odot\star, \tau_3\odot\star\}$	✓	✓	×

Table 1. Reaction enabling examples

Enabled reaction set. *Given a reaction R , the set of enabled subreactions by a conversation type $t\odot\tau$ is defined as*

$$R_{t\odot\tau} = \{R'.R \equiv R'|R'' \wedge \vdash R' : \mathbb{T} \wedge t\odot\tau \simeq \mathbb{T}\}$$

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

Table 2. Enabled reaction set example

Examples exploiting the *enabled reaction set* are given in Table 2, where $R_1 \equiv x : \tau_1 \odot \tau \rightarrow B_1$ $R_2 \equiv x : \tau_2 \odot \lambda \tau' \rightarrow B_2$. Notice that in the second case only one reaction ($R_1 | R_2$) is enabled. Upon the reception of a signal having conversation type T , both subreactions R_1 and R_2 will be concurrently activated. Notice also that in the fourth case 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 $t \odot \tau$ be a session conversation type. The set of preferred reactions in R with respect to $t \odot \tau$ is defined as:

$$R_{t \odot \tau \downarrow} = \left\{ R_1 \in R_{t \odot \tau} . \vdash R_1 : \mathbb{T}_1 \Rightarrow \forall R_2 \in R_{t \odot \tau} . \vdash R_2 : \mathbb{T}_2 \Rightarrow \left(\begin{array}{c} \mathbb{T}_1^+ \equiv \tau \\ \vee \\ \mathbb{T}_2^\times \subseteq \mathbb{T}_1^\times \end{array} \right) \right\}$$

Basically, each reaction $R_1 \in R_{t \odot \tau \downarrow}$ is composed only by check reactions for the τ session or, if it is composed only by lambda reactions, then it cannot exist 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:

$$E \triangleq e[\nu \tau . \bar{s} : \tau_1 + \tau_2 \odot \tau . 0]_{\tau_1 \rightsquigarrow c_1 | \tau_2 \rightsquigarrow c_2}^0$$

5 Operational Semantics

The operational semantics of XSC is given in the classical reduction style. The structural congruence over reactions, flows, behaviors and networks has been defined in Section 3.1. We define some auxiliary functions acting on flows and reactions that will be used in Section 5.1 to introduce the reduction relation over networks.

The *flow projection*, $(F) \downarrow_t$, takes a flow and a topic and yields the set of target component names for topic t . The flow projection is inductively defined below:

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

The *reaction projection*, $(R)\downarrow_{s:T}$, takes a reaction R and a signal s typed by T and returns a pair (B, R') . The first component, B , is the instantiated behavior (to which it is applied the variable binding $\{s/x\}$), the second component, R' , is the reaction to be installed. This permits to consume check reactions and to keep unchanged lambda reactions. Notice that the reaction projection is applied, by construction, to reactions that can consume the signal s . This assumption is guaranteed by the reduction rules, defined in Section 5.1, using the type system. The reaction projection is defined as follows:

$$\begin{aligned} (0)\downarrow_{s:\star} &= (0, 0) \\ (x : \tau' \odot \tau'' \rightarrow B)\downarrow_{s:t \odot \tau} &= (\{s/x\}B, 0) \\ (x : \tau' \odot \lambda \tau'' \rightarrow B)\downarrow_{s:t \odot \tau} &= (\{s/x, \tau/\tau''\}B, x : \tau' \odot \lambda \tau'' \rightarrow B) \\ \frac{(R_1)\downarrow_{s:t \odot \tau} &= (B', R') \\ (R_2)\downarrow_{s:t \odot \tau} &= (B'', R'')}{(R_1|R_2)\downarrow_{s:t \odot \tau} &= (B' | B'', R'|R'')} \end{aligned}$$

5.1 Reduction Rules

The reduction relation over networks (\rightarrow) is defined by the rules in Figure 5. Reactions can be added to a component by executing the behavioral primi-

$$\begin{aligned} \frac{}{a[+[x : \tau \odot \lambda \tau' \rightarrow B].B' | B'']_F^R \rightarrow a[B' | B'']_F^{R|x:\tau \odot \lambda \tau' \rightarrow B}} \quad (RLambdaUpd) \\ \frac{}{a[+[x : \tau \odot \tau' \rightarrow B].B' | B'']_F^R \rightarrow a[B' | B'']_F^{R|x:\tau \odot \tau' \rightarrow B}} \quad (RCheckUpd) \\ \frac{}{a[+[\tau \rightsquigarrow b].B | B']_F^R \rightarrow a[B | B']_F^{R|\tau \rightsquigarrow b}} \quad (FlowUpd) \\ \frac{(F)\downarrow_{t \odot \tau} = \vec{b}}{a[\bar{s} : t \odot \tau.B]_F^R \rightarrow a[B]_F^R \parallel \Sigma_{c_i \in \vec{b}} \langle s : t \odot \tau @ c_i \rangle} \quad (Emit) \\ \frac{\begin{array}{l} R \equiv R'|R_0 \\ R' \in R_{t \odot \tau \downarrow} \\ (R')\downarrow_{s:t \odot \tau} = (B', R'') \end{array}}{\langle s : t \odot \tau @ a \rangle \parallel a[B]_F^R \rightarrow a[B|B']_F^{R_0|R''}} \quad (RActivation) \\ \frac{N \rightarrow N'}{N \parallel N_1 \rightarrow N' \parallel N_1} \quad (NStep) \end{aligned}$$

Fig. 5. Operational Semantics

tives *RLambdaUpd* and *RCheckUpd*. These primitives change the a interface 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 the way a notification is dispatched. In particular, at emission time, the component a spawns into the network a signal targeted to all the components ($c_i \in b$) able to handle it, according to the $(F)\downarrow_t$ projection function previously defined. Once a signal envelop has been spawn into the network the *RActivation* primitive is executed on the target component activating, non deterministically, a reaction among the ones activated by the rule $R' \in R_{t\otimes\tau\downarrow}$. The activated reaction is so removed from the a interface by installing the 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 multiparty sessions. A typical scenario in which several agents are involved into the same session is represented by user-centric digital identity systems. In the following, we consider an application of the OpenID protocol, an open framework for distributed identity management. Obviously the solution presented can be easily adapted for similar systems (i-Name [1] and Microsoft CardSpace [2] to cite a few). 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 identifies uniquely a website. To reach this goal these systems define a special kind of services called Identity Providers that act as intermediate agents between the user and the service. Another key feature offered by OpenID is the decentralization of the authentication protocol decoupling the service from a particular identity provider. The protocol can be split into two parts. In the first phase, the user accesses its identity provider to make the authentication thus establishing a private session. The second phase regards the user access to a service supplying both the user identification and the identity provider that guarantees his credentials. Notice that the authentication mechanism adopted to identify an user on a Identity Provider is out of the specification of OpenID, and so it will not be threatened. In the protocol, here we only deal about the message exchange path followed by the involved parties.

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

1. The user initiates authentication with the identity provider by presenting its credentials.
2. The identity provider verifies user credentials and generate a new session shared with the user. The session will be used to identify the user.
3. The end user initiates authentication (Initiation) by presenting a User-Supplied Identifier to the Service Provider via their User-Agent.
4. The Service Provider establishes an Endpoint URL that the end user uses for authentication.

5. The Service Provider redirects the end user's User-Agent to the Identity Provider with an authentication request.
6. The Identity Provider establishes whether the end user is authorized to perform authentication and wishes to do so. Notice that the ways end user authenticates to its Identity Provider and any policies surrounding such authentication are out of scope for OpenID.
7. The Identity Provider redirects the end user's User-Agent back to the Service Provider with either an assertion stating that the authentication is approved or a message that authentication failed.
8. The Service Provider verifies the information received from the Identity Provider.

The OpenID protocol can be formally specified in XSC (see Figure 6). Notice that we omit to model the data exchanged among components, because we focus on the session exchanges and message sequences.

$$\begin{array}{l}
C \parallel IP \parallel SP \\
\\
C \triangleq \\
B_c \triangleq \\
B_{AuthOK}(s_{ip}) \triangleq \\
B_{Redirect_{si}}(s_{ip}, s_{sp}) \triangleq \\
B_{Redirect_{is}}(s_{ip}, s_{sp}, s_3) \triangleq \\
\\
IP \triangleq \\
R_{ip} \triangleq \\
B_{Auth}(r) \triangleq \\
B_{Delegate}(s_{ip}) \triangleq \\
B_{Verify} \triangleq \\
\\
SP \triangleq \\
R_{sp} \triangleq \\
B_{Claim}(r) \triangleq \\
B_{Check}(s_3) \triangleq \\
B_{Verified} \triangleq
\end{array}
\begin{array}{l}
c[B_c]_{Auth \rightsquigarrow i | Claim \rightsquigarrow s | Delegate \rightsquigarrow i} \\
\nu r. + [x : AuthOK \odot \lambda s_{ip} \rightarrow B_{AuthOK}(s_{ip})] \\
.\underline{credentials} : Auth \odot r.0 \\
\nu r. + [x : Redirect_{sp} \odot \lambda s_{sp} \rightarrow B_{Redirect_{si}}(s_{ip}, s_{sp})] \\
.\underline{identifier} : Claim \odot r.0 \\
+ [x : Redirect_{si} \odot \lambda s_3 \rightarrow B_{Redirect_{is}}(s_{ip}, s_{sp}, s_3)]. \\
\bar{x} : Delegate \odot s_{ip}.0 \\
+ [s_{sp} \odot \star \rightsquigarrow s]. \dots : s_{sp} \odot s_3.0 \\
\\
i[0]_{AuthOK \rightsquigarrow c | Redirect_{ip} \rightsquigarrow c | Verified \rightsquigarrow s} \\
x : Auth \odot \lambda r \rightarrow B_{Auth}(r) \\
\nu s_{ip}. + [x : Delegate \odot s_{ip} \rightarrow B_{Delegate}(s_{ip})]. \\
\bar{x} : AuthOK \odot s_{ip}.0 \\
\nu s_3. + [x : Verify \odot s_3 \rightarrow B_{Verify}]. \bar{x} : Redirect_{ip} \odot s_3.0 \\
\bar{x} : Verified \odot s_3.0 \\
\\
s[0]_{Redirect_{si} \rightsquigarrow c | Verify \rightsquigarrow i} \\
x : Claim \odot \lambda r \rightarrow B_{Claim}(r) \\
\nu s_{sp}. + [x : s_{sp} \odot \lambda s_3 \rightarrow B_{Check}(s_3)]. \bar{x} : Redirect_{si} \odot s_{sp}.0 \\
+ [x : Verified \odot s_3 \rightarrow B_{Verified}]. \bar{x} : Verify \odot s_3.0 \\
B
\end{array}$$

Fig. 6. XSC specification of the OpenID protocol

1. The user sends its credentials to the Identity Provider (B_c), rising an *Auth* event. Notice that the client creates a new reaction to receive an event cor-

- responding to the successful authentication ($AuthOK$) from the identity provider.
2. 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 credential. The service provider raises a successful authentication event ($AuthOK$), communicating the generated session. Notice that we assume that the user always succeeds its authentication: therefore we do not model the implementation verification of the user credentials (x). 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.
 3. 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 raising a $Claim$ event.
 4. When a service provider receives a $Claim$ event, it delegates the authentication of the identity (x) to the identity provider. This is performed redirecting the client to the identity provider. Notice that the service provider generates a new session (s_{sp}) that is communicated via the redirect request ($\bar{x} : Redirect_{si} \odot s_{sp}.0$). The generated session is used as a new event, the service provider waits this event to perform the authentication. In Open-ID this is implemented through the generation of a user-specific url.
 5. 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 to them the authentication.
 6. When the identity provider receives a delegate event for the authenticated user ($Delegate \odot s_{ip}$) it generates a three-party session (s_3) and requests to communicate this to the service provider through a redirect response to the user. The identity provider and the service provider use this session to verify the user claims.

7 Concluding Remarks

We introduced a process algebra to handle multiparty sessions and coordination in an EN framework. Our approach is based on a dynamic types that naturally support and extend typed-based EN systems.

We exemplified the features of the approach by modelling the OpenID protocol. In the scope of our future work there is an encoding Global Calculus (GC), the language proposed in [20], in XSC. A beneficial outcome of such research would be the possibility of studying properties of abstract protocols (like OpenID) at the higher of abstraction than XSC and then correctness of implementation is achieved by simply compiling the GC formalizations into XSC.

Additionally, as discussed in [15], we have provided a middleware implemented in Java, called Java Signal Core Layer (JSCL), where the basic primitives

of SC are realised. A part of the new concepts introduced in XSC have already been implemented into JSCL (e.g. logical ports and signal sessions), while topic creation and structured topic composition are under development.

We plan to provide different interpretation for the algebraic structure of topics. For instance, by relaxing idempotency of \times we get a theory which allows one to *count* the number of structured topics.

Finally, we remark that the type system described in this paper yields a *constraint semiring* structure that has been successfully exploited to model QoS aspects of distributed systems [6, 8, 4, 12, 16]. We argue that this will allow us to express QoS driven coordination policy within our type system.

References

1. i-name specifications. Technical report. Available at "<http://www.inames.net/developers.html>".
2. Microsoft cardspace. Technical report. Available at "<http://msdn2.microsoft.com/en-us/netframework/aa663320.aspx>".
3. Openid specifications. Technical report. Available at <http://openid.net/specs.bml>.
4. S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *IJCAI (1)*, pages 624–630, 1995.
5. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
6. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. In D. L. Métyer, editor, *ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2002.
7. M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. Scc: A service centered calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.
8. M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. *ESOP'07. 16th European Symposium on Programming*, 2007. To appear.
9. A. Carzaniga, D. Rosenblum, and L. Alexander. Achieving scalability and expressiveness in an internet-scale event notification service. pages 219–227, 2000.
10. A. Carzaniga and A. 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, pages 59–68, London, UK, 2002.
11. A. Carzaniga and A. Wolf. Forwarding in a content-based network. In A. Feldmann, M. Zitterbart, J. Crowcroft, and D. 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, 2003.
12. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Basic Calculus for Modelling Service Level Agreements. In J. Jacquet and G. Picco, editors, *International Conference on Coordination Models and Languages*, volume 3454, pages 33 – 48, April 2005.

13. P. Eugster and R. Guerraoui. Distributed programming with typed events. *IEEE Software*, 21(2):56–64, Mar./Apr. 2004.
14. G. L. Ferrari, R. Guanciale, and D. Strollo. Event based service coordination over dynamic and heterogeneous networks. In A. Dan and W. Lamersdorf, editors, *IC-SOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 453–458. Springer, 2006.
15. G. L. Ferrari, R. Guanciale, and D. Strollo. Jscl: A middleware for service coordination. In E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2006.
16. D. Hirsch and E. Tuosto. SHReQ: A Framework for Coordinating Application Level QoS. In K. Bernhard and B. Bernhard, editors, *3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 425–434, 2005.
17. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. *Lecture Notes in Computer Science*, 1381:122–141, 1998.
18. Y. Huang and D. Gannon. A comparative study of web services-based event notification specifications. In *ICPP Workshops*, pages 7–14. IEEE Computer Society, 2006.
19. Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Department, Indiana University, 2003.
20. K. H. Marco Carbone and N. Yoshida. Structured communication-centred programming for web services. *ESOP'07. 16th European Symposium on Programming*, 2007. To appear.
21. D. Tam, R. Azimi, and H.-A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In K. Aberer, V. Kalogeraki, and M. Koubarakis, editors, *Databases, Information Systems, and Peer-to-Peer Computing*, volume 2944, pages 138–152, 2003.

A Free names

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

$$\begin{aligned}
fn(0) &= \emptyset \\
fn(!B) &= fn(B) \\
fn(+[\tau \rightsquigarrow a].B') &= \{\tau, a\} \cup fn(B') \\
fn(B_1 \mid B_2) &= fn(B_1) \cup fn(B_2) \\
fn(+[x : \tau \odot \tau' \rightarrow B].B') &= fn(B) \setminus \{x\} \cup \{\tau, \tau'\} \cup fn(B') \\
fn(+[x : \tau \odot \lambda \tau' \rightarrow B].B') &= fn(B) \setminus \{x, \tau'\} \cup \{\tau\} \cup fn(B') \\
fn(\bar{s} : t \odot \tau.B') &= fn(B') \cup \{s, \tau\} \cup fn(t) \\
fn(\nu \tau.B') &= fn(B') \setminus \{\tau\} \\
\\
fn(0) &= \emptyset \\
fn(R_1 \mid R_2) &= fn(R_1) \cup fn(R_2) \\
fn(x : \tau \odot \tau' \rightarrow B) &= fn(B) \setminus \{x\} \cup \{\tau, \tau'\} \\
fn(x : \tau \odot \lambda \tau' \rightarrow B) &= fn(B) \setminus \{x, \tau'\} \cup \{\tau\} \\
fn(0) &= \emptyset \\
fn(F_1 \mid F_2) &= fn(F_1) \cup fn(F_2) \\
fn(\tau \rightsquigarrow b) &= \{\tau, b\} \\
\\
fn(\emptyset) &= \emptyset \\
fn(\nu \tau.N) &= fn(N) \setminus \{\tau\} \\
fn(\langle s : t \odot \tau @ a \rangle) &= \{s, a, \tau\} \cup fn(t) \\
fn(N_1 \parallel N_2) &= fn(N_1) \cup fn(N_2) \\
fn(a[B]_F^R) &= fn(B) \cup fn(F) \cup fn(R) \cup \{a\} \\
\\
fn(\tau) &= \{\tau\} \\
fn(\varepsilon) = fn(\star) &= \emptyset \\
fn(t_1 \odot \tau) &= fn(t_1) \cup \{\tau\} \\
fn(t_1 \odot \star) &= fn(t_1) \\
fn(t_1 \times t_2) = fn(t_1 + t_2) &= fn(t_1) \cup fn(t_2)
\end{aligned}$$