# JSCL: a Middleware for Service Coordination*

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

[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

**Abstract.** This paper describes the design and the prototype implementation of a middleware, called Java Signal Core Layer (JSCL), for coordinating distributed services. JSCL supports the coordination of distributed services by exploiting an *event notification* paradigm. The design and the implementation of JSCL has been inspired and driven by its formal specification given as a process calculus, the Signal Calculus (SC). At the experimental level JSCL has been exploited to implement Long Running Transactions (LRTs).

## 1 Introduction

One important challenge of the Software Engineering field is represented by the so called Service Oriented Architectures (SOAs) [**?** ]. In the SOA approach applications are developed by coordinating the behavior of autonomous components distributed over an overlay network. Middleware for coordinating services are extremely important to the success of SOAs. Several research and implementation efforts are currently devoted to design and to implement middleware for coordinating distributes services (see ORC [**?** ], BPEL [**?** ], WS-CDL [**?** ] and SIENA [**?** ] to cite a few). However, research is still underway. The aim of this paper is to contribute to this theme of research by developing a middleware for coordinating services based upon a formal basis. The strict integration between theory and practice is the key feature of our proposal. In particular, this paper describes the design and the prototype implementation of the JSCL middleware. At the abstract level JSCL takes the form of a process calculus, SC, a dialect of the Ambient Calculus [**?** ] with asynchronous communication facilities. At the implementation level, JSCL takes the form of a collection of Java APIs.

The starting point of our work is the event-notification paradigm. We assume to coordinate service behaviors through the exchange of (typed) signals. The basic building blocks of our middleware are called *components*. A component represents a "simple" service interacting though a signal passing mechanism. Components are basic computational units performing some internal operations and can be composed and distributed over a network. Composition of components, yields a new one that can be used in further compositions. Each component is identified by an unique *name*, which, intuitively,

* Research partially supported by the EU, within the FET GC Project IST-2005-16004 Sensoria, and FIRB Project TOCAI.IT.

can be through as the URI of the published service. In this paper we assume as given the set of names of the components involved into a system with no assumption on the mechanisms adopted to retrieve them (e.g. UDDI service directories, registries, etc.).

The signals exchanged among components are basically 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, in the literature (e.g. [**?** ]), with the term *topic*. Hence components are *reactive blocks* that declare the subset of signals they are interested in together with their *reactions* upon event notifications. The reactions are modeled by associating functional modules to topics of received signals. Once a signal of a well defined topic is received, the proper reaction is activated.

The way the events are notified to the subscribed components is strictly related to the specific coordination pattern chosen. Different *conversational styles* can be adopted to implement the way the participants are involved into a coordination, mainly split into two main groups: *orchestration* and *choreography* (as discussed in [**?** ]). Briefly, the first solution defines an intermediate agent, the orchestrator, that is responsible to decide, at each step, which are the actions that must be performed by each component. The choreography, instead, identifies a more distributed scenario in which, each participant is responsible for its moves and the whole work-flow is executed following a pre-defined plan. Basically, the orchestration suggests a centralization point that is responsible for implementing the subscriptions and the notification forwarding. Such solution is closely related to the ideas of tuple space based systems and brokered event notification. Using the choreography, instead, each component can act both as publisher or subscriber for other components and the delivering of signals is implemented through *peer-to-peer* like structures. In this paper, we adopt the choreography approach since it better fits with the signal passing paradigm.

This paper is organized as follows. Section 2 introduces the Signal Calculus (`SC`). `SC` is a calculus for describing coordination primitives for components interacting through a signal passing mechanism. Section 3 describes `JSCL` APIs and the way components can be programmed. Basically, `JSCL` is a *ligthweight* framework for modeling distributed services by composing components that use signals for notifying events to other interested components in the style of `SC`. In section 4, as a case study, we describe the usage of `JSCL` as programming middleware for Long Running Transactions (LRTs) [**?** ].

## 2 Signal Calculus: SC

The Signal Calculus (`SC`) is a process calculus in the style of [**? ?** ] specifically designed to describe coordination policies of services distributed over a network. `SC` describes computation via the choreography of local service behavior. In this section, we present the syntax and the operational semantics of `SC`.

### 2.1 SC Syntax

The main concepts of `SC` are signals, components, reactions, flows and networks. The data carried by a *signal* are the signal name and the conversation schema. A *signal*

*name* represents an identifier of the current conversation (e.g. the session-id) and a *conversation schema* represents the kind of event (e.g. onMouseOver). New signals can be sent either by autonomous components or as reaction to other signals. In this paper, we present SC focusing only on the primitives needed to design coordination protocols. Hence, operations on conversation schemata are not defined, since they can be expressed at a higher level detail of abstraction. Of course, SC can be extended by adding types for conversation schemata (e.g. in the form of XML Schema) [**? ? ? ?** ]. A SC *component* is a wrapper for a *behavior*. Intuitively a SC component represents an autonomous service available over a network. Each SC component is uniquely identified by its name and contains a local behavior and an interface. Components can behave either as signal emitters or as signal handlers. Similarly to the event-notification pattern, signal handlers are associated to signals and are responsible for their management. The SC component *interface* is structured into *reactions* and *flows*. Reactions describe component behavior and the action of variable binding upon signal reception. Indeed, the reception of a signal acts like a trigger that activates the execution of a new behavior within the component.

Orchestration among components is implemented through flows. Flows represent the local view (component view) of the choreography, that is the set of local communications that have to be performed to satisfy the choreography demands. Each component flow declares the associations among signals, the conversation schema and the set of handlers. The connections among components are strictly related to a particular conversation schema thus offering the possibility to express different topologies of connectivity, depending on the schema of the outgoing signals. Both component reactions and flows are programmable, and they can be dynamically modified by the components.

Components are structured to build a *network* of services. A network provides the facility to transport envelopes containing the signals exchanged among components.

We now introduce the main syntactic categories of our calculus together with some notational machineries. We assume a finite set of conversation schemata ranged by $\tau_1, ..., \tau_k$, a finite set of component names ranged by $a, b, c...$ and a finite set of signal names ranged by $s_1, s_2, ....$ We also assume a set *Var* of variable names whose typical elements are $x, y, z....$ We use $\vec{a}$ to denote a set of names $a_1, ..., a_n$. Finally, we use $\sigma$ to denote a substitution from variable names to signal names.

Reactions ($R$) are described by the following grammar:

$$
\begin{array}{lll}
(\textsc{Reactions})\ R ::= & 0 & \text{Nil} \\
& *(x : \tau \rightarrow B) & \text{Unit reaction} \\
& R|R & \text{Composition}
\end{array}
$$

A reaction is a set (possibly empty) of unit reactions. A unit reaction $*(x : \tau \rightarrow B)$ triggers the execution of the behavior $B$ upon reception of a signal tagged by the schema $\tau$. Notice that $x : \tau$ acts as a binder for the variable $x$ within the behavior $B$. The syntax of behaviors will be given below. As usual we assume to work up-to alpha-conversion. Free and bound names are defined in the standard way.

Flows ($F$) are described by the following grammar:

$$\begin{array}{lll} \text{(FLOWS)}\ F ::= & 0 & \text{Nil} \\ & \tau \triangleright \vec{a} & \text{Unit flow} \\ & F \bullet F & \text{Composition} \end{array}$$

A flow is a set (possibly empty) of unit flows. A unit flow $\tau \triangleright \vec{a}$ describes the set of component names $\vec{a}$ where outgoing signals having $\tau$ as conversation schema have to be delivered.

Reactions and flows are defined up-to a structural congruence ($\equiv$). Indeed, we assume that $\bullet$ and $|$ are associative, commutative and with 0 behaving as identity. Notice that such equations allow us to freely rearrange reactions and flows.
We define two auxiliary *schema* functions $S(R)$ and $S(F)$ that, respectively, return the set of conversation schemata on which the reaction $R$ and the flow $F$ are defined.

$$\begin{array}{ll} S(0) & = \emptyset \\ S(*(x:\tau \rightarrow B)) & = \{\tau\} \\ S(R_1|R_2) & = S(R_1) \cup S(R_2) \end{array} \qquad \begin{array}{ll} S(0) & = \emptyset \\ S(\tau \triangleright \vec{a}) & = \{\tau\} \\ S(F_1|F_2) & = S(F_1) \cup S(F_2) \end{array}$$

We say that a reaction is *well-formed* (and we write $R\checkmark$) if there is no overlay among the conversation schemata triggered. The notion of well-formed reaction is inductively defined below.

$$\frac{}{0\checkmark} \qquad \frac{}{*(x:\tau \rightarrow B)\checkmark} \qquad \frac{R_1\checkmark \quad R_2\checkmark \quad S(R_1) \cap S(R_2) \equiv \emptyset}{(R_1|R_2)\checkmark}$$

We also introduce two *projection* functions $R \downarrow_{s:\tau}$ and $F \downarrow_\tau$; the first takes a well-formed reaction $R$ and a signal $s$ of schema $\tau$ and returns a pair consisting of the variable substitution and the activated behavior. The second projection takes a flow $F$ and a schema $\tau$ and returns the set of component names linked to the flow having schema $\tau$. The two projections are defined below.

$$\begin{array}{lll} 0 \downarrow_{s:\tau} & = (\{\}, 0) & \\ (*(x:\tau \rightarrow B)|R) \downarrow_{s:\tau} & = (\{s/x\}, B) & \\ (*(x:\tau_1 \rightarrow B)|R) \downarrow_{s:\tau} & = R \downarrow_{s:\tau} & if\ \tau_1 \neq \tau \end{array} \qquad \begin{array}{lll} 0 \downarrow_\tau & = \{\} & \\ (\tau \triangleright \vec{a} \bullet F) \downarrow_\tau & = \vec{a} \cup (F \downarrow_\tau) & \\ (\tau_1 \triangleright \vec{a} \bullet F) \downarrow_\tau & = F \downarrow_\tau & if\ \tau_1 \neq \tau \end{array}$$

Finally we say that a flow is well-formed (and we write $F\checkmark$) if there is no overlay among the linked components for all conversation schemata. The notion of well-formed flow is inductively defined below.

$$\frac{}{0\checkmark} \qquad \frac{}{(\tau \triangleright \vec{a})\checkmark} \qquad \frac{F_1\checkmark \quad F_2\checkmark \quad \forall_{\tau \in (S(F_1) \cup S(F_2))} (F_1 \downarrow_\tau \cap F_2 \downarrow_\tau \equiv \emptyset)}{(F_1 \bullet F_2)\checkmark}$$

Hereafter, we assume that reactions and flows are always well-formed.

Component behaviors ($B$) are defined by the following grammar:

| | | |
|---|---|---|
| (BEHAVIORS) $B ::= 0$ | | Nil |
| | $+R[x:\tau \rightarrow B]$ | Reaction update |
| | $+F[\tau \triangleright \vec{a}]$ | Flow update |
| | $\bar{s}:\tau.B$ | Asynchronous signal emission |
| | $B \mid B$ | Parallel composition |
| | $!B$ | Bang |

A reaction update $+R[x:\tau \rightarrow B]$ extends the reaction part of the component interface, providing the ability to react to a signal of schema $\tau$ activating the behavior $B$. Such operation ensures that the resulting reaction is well-formed and permits to dynamically change the reaction interface. Similarly, a flow update $+F[\tau \triangleright \vec{a}]$ extends the component flows, appending the component names in $\vec{a}$ to the set of component names to which the signals of schema $\tau$ are delivered. An asynchronous signal emission $\bar{s}:\tau.B$ first spawns into the network a set of envelopes containing the signal $s$, one for each component name declared in the flow having schema $\tau$, and then activates $B$. As usual, the bang replication $!B$ represents a behavior that can always activate a new copy of the behavior $B$. When it is clear from the context, we will omit the Nil behavior, writing $\bar{s}:\tau$ for $\bar{s}:\tau.0$ and $B$ for $B|0$.

Networks ($N$) are defined by the following grammar:

| | | |
|---|---|---|
| (NETWORKS) $N ::= \emptyset$ | | Empty net |
| | $a[B]_F^R$ | Component |
| | $N \| N$ | Parallel composition |
| | $< s:\tau@a >$ | Signal envelope |

A component $a[B]_F^R$ describes a component of name $a$ with behavior $B$, reaction $R$ and flow $F$. A signal envelope $< s:\tau@a >$ describes a message containing the signal $s$ of schema $\tau$ whose target component is the component named $a$. We use $\sum_{x \in \vec{a}} B$ to denote the parallel composition of $B\{n/x\}$ for each name $n$ in the set $\vec{a}$. A SC component is closely related to the notion of ambient [**?** ] as it describes a behavior wrapped within a named context. Differently from the ambient calculus, SC networks are flat, that means there is no hierarchy of components.

**Examples** To better present how the basic SC concepts can be used to model service coordination we introduce a simple example. Suppose to have a producer $p$ and a consumer $c$ both accessing a shared data space. We assume a synchronization policy, namely a consumer can get its resource only after a producer has produced it. The problem can be modeled as displayed in Figure 1. $P$ starts its execution performing the (internal) behavior $B_p$ that modifies the state of the data space that has to be read by $C$. When the data have been modified, $B_p$ executes a signal emission of a signal of schema *produced* ($\bar{s}:$ *produced*) in order to inform $C$ that the required resources are now available. Upon notification, $C$ automatically starts and takes the resource in the data space

performing its internal behavior $B_c$. We assume that $B_c$ executes a signal emission of a signal of schema *consumed* ($\bar{x} : consumed$) in order to inform $P$ that it can produce a new resource. Notice that the name of the signal emitted is the same of the signal received. Moreover $C$ is not a running process, it is an idle entity that is activated only at signal reception.

$$P \triangleq p[\bar{s} : produced]^{x:consumed \to \bar{x}:produced}_{produced \triangleright c} \qquad C \triangleq c[0]^{x:produced \to \bar{x}:consumed}_{consumed \triangleright p} \qquad Net \triangleq P \| C$$

**Fig. 1.** Components $p$ and $c$ share a data space

In the previous example, we presented two components with a statically defined choreography. However the producer and the consumer can be dynamically linked together (e.g. at the start up phase) using reaction update and flow update, thus providing a dynamic choreography scenario. This example is expressed in SC through the components and the network defined in Figure 2. Notice that, since component name passing has not been modeled in SC, we assume each component knows the names of the externally published components. We can enrich the SC core providing component name communication, thus yielding a true dynamic choreography in the style of the $\pi$-calculus [**?** ].

$$\begin{aligned} P &\triangleq p[+F[produced \triangleright c]| + R[x : consumed \to \bar{x} : produced.0]|\bar{s} : produced]^0_0 \\ C &\triangleq c[+F[consumed \triangleright p]| + R[x : produced \to \bar{x} : consumed.0]]^0_0 \\ Net &\triangleq P \| C \end{aligned}$$

**Fig. 2.** Components $p$ and $c$ share a data space

### 2.2 SC Semantics

SC semantics is defined in a reduction style [**?** ]. We first introduce a structural congruence over behaviors and networks. The structural congruence for component behaviors ($\equiv_B$) is defined by the following rules:

$$B_1|B_2 \equiv_B B_2|B_1 \qquad (B_1|B_2)|B_3 \equiv_B B_1|(B_2|B_3) \qquad \emptyset|B \equiv_B B \qquad !B \equiv_B B|!B$$

As usual the bang operator allows us to express recursive behaviors.
Structural congruence for networks $\equiv_N$ is defined by the following rules:

$$N\|M \equiv_N M\|N \qquad (M\|N)\|O \equiv_N M\|(N\|O) \qquad \emptyset\|N \equiv_N N$$

$$a[\emptyset]^0_F \equiv_N \emptyset \qquad \frac{F^1 \equiv F^2 \;\; R^1 \equiv R^2 \;\; B^1 \equiv_B B^2}{a[B^1]^{R^1}_{F^1} \equiv_N a[B^2]^{R^2}_{F^2}}$$

A component having *nil* behavior and empty reaction can be considered as the empty network since it has no internal active behavior and cannot activate any behavior upon reception of a signal. Two components are considered structurally congruent if their internal behaviors, reactions and flows are structurally congruent. When it is clear from the context, we will use the symbol $\equiv$ for both $\equiv_B$ and $\equiv_N$.

The reduction relation of networks ($\rightarrow$) is defined by the rules in Figure 3. The rule (RUPD) extends the component reactions with a further unit reaction (the parameter of the primitive). The rule requires that the resulting reaction is well-formed. The rule (FUPD) extends the component flows with a unit flow. Also in this case a well-formed resulting flow is required. The rule (OUT) first takes the set of component names $\vec{a}$ that are linked to the component for the conversation schema $\tau$ and then spawns into the network an envelope for each component name in the set. The rule (IN) allows a signal envelope to react with the component whose name is specified inside the envelope. Notice that signal emission rule (OUT) and signal receiving rule (IN) do not consume, respectively, the flow and the reaction of the component. This feature provides SC with a further form of recursion behavior. The rules (STRUCT) and (PAR) are standard rules. In the following, we use $N \rightarrow^{+} N_1$ to represent a network $N$ that is reduced to $N_1$ after a finite number of steps.

$$\frac{R|*(x:\tau \rightarrow B)\checkmark}{a[+R[x:\tau \rightarrow B]|Q]^R_F \rightarrow a[Q]^{R|*(x:\tau \rightarrow B)}_F}\ (RUPD) \qquad \frac{F \bullet \tau \triangleright \vec{a}\checkmark}{a[+F[\tau \triangleright \vec{a}]|Q]^F_R \rightarrow a[Q]^R_{F \bullet \tau \triangleright \vec{a}}}\ (FUPD)$$

$$\frac{F \downarrow_\tau = \vec{a}}{a[\bar{s}:\tau.P|Q]^R_F \rightarrow a[P|Q]^R_F|\sum_{a_i \in \vec{a}} < s:\tau@a_i >}\ (OUT) \qquad \frac{R \downarrow_{s:\tau} = (\sigma, B)}{< s:\tau@a > |a[Q]^R_F \rightarrow a[Q|\sigma B]^R_F}\ (IN)$$

$$\frac{N \equiv N_1 \rightarrow M_2 \equiv N_3}{N \rightarrow N_3}\ (STRUCT) \qquad \frac{N \rightarrow N_1}{N|N_2 \rightarrow N_1|N_2}\ (PAR)$$

**Fig. 3.** Semantic Rules

**Examples** To describe how SC semantics rules work, we provide a short description of the execution of the examples given in Figure 1 and 2. As a shorthand, we write $\tau_p$ for the conversation schema *produced* and $\tau_c$ for *consumed*. The network in Figure 1 contains only one active component; namely the producer $p$ emits the signal, spawning into the network an envelope for the consumer $c$. This is represented by:

$$\frac{(\tau_p \triangleright c) \downarrow_{\tau_p} = c}{p[\bar{s}:\tau_p]^{(x:\tau_c \rightarrow \bar{x}:\tau_p)}_{(\tau_p \triangleright c)} \rightarrow p[0]^{(x:\tau_c \rightarrow \bar{x}:\tau_p)}_{(\tau_p \triangleright c)}| < s:\tau_p@c >}\ (OUT)$$

The envelope reacts with the consumer component, activating inside the component the behavior of the corresponding reaction:

$$\frac{(x:\tau_p \to \bar{x}:\tau_c)\downarrow_{s:\tau_p} = (\{s/x\}, \bar{x}:\tau_c)}{< s:\tau_p @ c > |\, c[0]^{(x:\tau_p \to \bar{x}:\tau_c)}_{(\tau_c \triangleright p)} \to c[\bar{s}:\tau_c]^{(x:\tau_p \to \bar{x}:\tau_c)}_{(\tau_c \triangleright p)}}\ (IN)$$

In a similarly way, the consumer component $c$ sends an envelope to the producer $p$, thus activating the proper internal behavior:

$$p[0]^{(x:\tau_c \to \bar{x}:\tau_p)}_{(\tau_p \triangleright c)} | c[\bar{s}:\tau_c]^{(x:\tau_p \to \bar{x}:\tau_c)}_{(\tau_c \triangleright p)} \to^+ p[\bar{s}:\tau_p]^{(x:\tau_c \to \bar{x}:\tau_p)}_{(\tau_p \triangleright c)} | c[0]^{(x:\tau_p \to \bar{x}:\tau_c)}_{(\tau_c \triangleright p)}$$

In the second example all the two components have active internal behaviors. The producer can update its flow by adding the link to the consumer for signals of schema $\tau_p$, as follows:

$$\frac{(0 \bullet \tau_p \triangleright c)\checkmark}{p[+F[\tau_p \triangleright c] \mid +R[x:\tau_c \to \bar{x}:\tau_p] \mid \bar{s}:\tau_p]^0_0 \ \to\ p[+R[x:\tau_c \to \bar{x}:\tau_p] \mid \bar{s}:\tau_p]^0_{(\tau_p \triangleright c)}}$$

Then we apply the reduction rule for the reaction update of the producer:

$$\frac{(0|x:\tau_c \to \bar{x}:\tau_p)\checkmark}{p[+R[x:\tau_c \to \bar{x}:\tau_p]|\bar{s}:\tau_p]^0_{(\tau_p \triangleright c)} \ \to\ p[\bar{s}:\tau_p]^{(x:\tau_c \to \bar{x}:\tau_p)}_{(\tau_p \triangleright c)}}$$

After these reductions the producer component has created a link to the consumer for signals of schema $\tau_p$ and can receive signals of schema $\tau_c$. In a similar way the consumer component updates its reaction and flow:

$$p[\bar{s}:\tau_p]^{(x:\tau_c \to \bar{x}:\tau_p)}_{(\tau_p \triangleright c)} | c[+F[\tau_c \triangleright p]| + R[x:\tau_p \to \bar{x}:\tau_c]]^0_0 \to^+ p[\bar{s}:\tau_p]^{(x:\tau_c \to \bar{x}:\tau_p)}_{(\tau_p \triangleright c)} | c[0]^{(x:\tau_p \to \bar{x}:\tau_c)}_{(\tau_c \triangleright p)}$$

## 3 Java Signal Core Layer

Java Signal Core Layer (JSCL) consists of a collection of Java API implementing the coordination primitives formally described by SC. In JSCL, known concepts of the event-based paradigm are considered in a distributed and open environment where components can be allocated on different execution sites and can join existing execution of other components. In the following, we often refer to components as *services* meaning that the current version of JSCL is specifically tailored to coordinate web services. However JSCL can be easily adapted to different technologies (e.g. CORBA). The essential ingredients of JSCL are *signals*, *components*, *signal links* and *input ports* corresponding, respectively, to the concepts of signals, components, flows and reactions defined in section 2. The notion of SC network is implemented by introducing an intermediate layer, the *Inter Object Communication Layer* (*IOCL*). The *IOCL* contains the set of primitives for creating, publishing, retrieving and connecting components. These operations are strictly related to the execution environment. To support multiple

definitions of *IOCL*, these primitives have been developed as *plugins* that can be directly accessed at run-time. Such layer has been introduced to make JSCL more flexible and allows the interoperability of different technologies for inter object communication like CORBA, Web Services (see [**?** ] for more details). In particular, the *IOCL* layer provides the mechanisms to implement the *data serialization* (e.g. SOAP message envelops for WSs, etc.) and the *deployment phase* (e.g. stub generation, dynamic proxy binding, etc.). Moreover each IOCL *plugin* defines the way components are identified by extending the basic interface ComponentAddress. The way components are named in JSCL strictly depends over the underlying overlay network adopted (e.g. an URL if the selected *iocl plugin* is based on WSs or CORBA, a couple $(IP, port)$ if sockets are used, an unique name if memory access is used etc.).

In JSCL, the set of information conveyed in each *signal* is split into two parts. The first contains information useful for coordination: the unique *name* of the signal instance and its *type*. The second part contains the payload of current request, the *session data*. Notice that the session data are not modeled in SC since the calculus only deals with the primitives needed for implementing the coordination among components. Signals are classified into *signal types* (*types* for short) that associate each signal to the class of events they belong to. Signal types are namely the SC conversation schemata. Signals have been modeled as *non persistent* entities; once the notification for an event has been delivered to all the interested handlers, the corresponding signal is *removed* from the system. This property is mandatory if we want to keep the distribution of the connections and their management. This feature, however, is not a limitation: persistent signals can be easily introduced in our middleware. Signals, in JSCL, are always sent in *non anonymous* way, meaning that it is always possible to know the sender of each signal. Such constraint is useful, at implementation level, if we want to extend the middleware with authoring primitives on the links.

The event notification mechanism of subscription is implemented through the creation of *input ports* and *signal links* connecting components. These operations correspond respectively to the primitives *reaction update* and *flow update* defined in SC.

An *input port* is a tuple $i = (sig_T, Task)$ meaning that the port can receive signals of type $sig_T$ whose handler is the process *Task*. Each component is able to specify only one input port for each $sig_T$. The notion of input port *i*, signal type $sig_T$ and handling task *Task* map respectively the reaction *R*, the conversation schema $\tau$ and the behavior *B* defined in SC. The component interface is obtained by taking the set of $sig_T$ for which there is a bound input port, and the set of $sig_T$ for which there is at least a signal link defined. Such sets in SC are given respectively by the auxiliary schema functions $S(R)$ and $S(F)$. Figure 4 shows a graphical notation for JSCL input ports.

Connections between two components are implemented through *signal links*. A signal link is a tuple $l = (sig_T, S, R)$ and represents an virtual channel among the signals of type $sig_T$ emitted by *S* and the handler *R*. Creating a new link between *S* and *R* requires that the input port corresponding to $sig_T$ has been previously created by *R* with the right permissions for *S*. Basically, signal links are the linguistic device of JSCL for subscription/notification. We already pointed out that several receivers can be linked with the same signal type and the same sender. All the signal links created are *well formed* conforming to the rules defined in section 2.1. Namely further creations of links

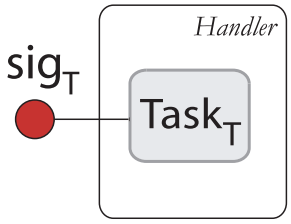


**Fig. 4.** JSCL input port

$(sig_T, S, R)$ are idempotent. The primitive for creating new signal links can be invoked outward the components by an external application that will connect all (or a subset of) the components among them and this will be the only agent conscious of the topology of the network. This assumption is useful to preserve the autonomy of the components from the particular system in which they are acting. The creation of a link is implemented through the *IOCL* component. Links in JSCL are *typed*, *unidirectional* and *"peer-to-peer"*. More complex scenarios (e.g., multi-casting, bi-directionality, etc.) can be obtained by introducing the suitable notions of links. For instance, multi-casting is achieved by connecting the same emitter to several handlers.

**Example** We now describe how the *producer/consumer* example modeled in SC in section 2.2 can be implemented by exploiting the JSCL APIs. This provides a basic idea of the programmability offered by the middleware. The Figure 5 displays the choreography between two components $P$ and $C$ representing respectively the services implementing a *producer* and a *consumer*. The topics of signals exchanged in the system are $sig_{prod}$ and $sig_{cons}$ corresponding to notifications for events *produced* and *consumed* which can be raised respectively from $P$ and $C$.

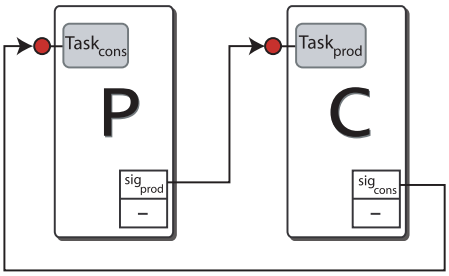


**Fig. 5.** JSCL: producer-consumer example.

The designing of the whole application can be logically split into three phases: *i)* the creation of the components, *ii)* the declaration of the reactions associated to the

components and *iii)* the publication of services and the designing of the connections. In the following, we assume to exploit the XSOAP [**?** ] IOCL *plugin*, and to use the `ServiceAddress`[3] class, essentially an URL, to define component names.

```
IOCLPluginLoader loader = new IOCLPluginLoader("jscl.core.IOCL.XSoap.IOCLImpl");
IOCLPluginI iocl = loader.getIOCLInstance();
ServiceAddress Paddress = new ServiceAddress ("http", "jordie", 9092, "", "Producer");
ServiceAddress Caddress = new ServiceAddress ("http", "jordie", 9092, "", "Consumer");
GenericComponent producer = iocl.createComponent(Paddress);
GenericComponent consumer = iocl.createComponent(Caddress);
```

**Code 1**: Producer & consumer creation.

The Code 1 illustrates the `JSCL` code for creating the needed services. First we instantiate a new `IOCLPlugin` that will be used to create the components. The creation of new components occurs by invoking the method *createComponent*, whose parameter is the address of the component itself. Alternatively, the *iocl* layer can be used to retrieve already published services by invoking the method *iocl.getComponent* which, given an address, returns a component proxy bound to it. Once our components have been built, we must program their reactions by binding them to new *input ports* as shown in Code 2. Roughly Code 2 describes the processes corresponding to the $Task_{cons}$ and $Task_{prod}$ depicted in Figure 5.

```
consumer.addInputPort(
 new SignalInputPort (SignalTypes.Sig_prod,
  new SignalHandlerTask(consumer){
   public Object handle (Signal signal){
    try{
     // Consumes the resource
     ...
     signal.setType(SignalTypes.Sig_cons);
     this.getParent().emitSignal(signal);
    } catch (GenericException e){
     e.printStackTrace();
    }
     return null;
   }
  }
 )
);
```

```
producer.addInputPort(
 new SignalInputPort (SignalTypes.Sig_cons,
  new SignalHandlerTask(producer){
   public Object handle (Signal signal){
    try{
     // Produces the resource
     ...
     signal.setType(SignalTypes.Sig_prod);
     this.getParent().emitSignal(signal);
    } catch (GenericException e){
     e.printStackTrace();
    }
     return null;
   }
  }
 )
);
```

**Code 2**: Binding of *input ports*.

The last step is the publication of the created services and the creation of links as shown in Code 3.

```
// Component publication
iocl.registerComponent (producer);
iocl.registerComponent (consumer);

// Creation of links
iocl.createLink (SignalTypes.Sig_prod, Paddress, Caddress);
iocl.createLink (SignalTypes.Sig_cons, Caddress, Paddress);
```

**Code 3**: Link creation.

[3] `ServiceAddress` is an implementation of `ComponentAddress` defined in section 3.

Once the orchestration has been declared, we can start its execution by the signal emission of the producer component, which is the only active internal behavior. Here, for simplicity, the primitives for creating and publishing the components depicted in Code 1 and in Code 3, for simplicity, are collapsed into an unique block, using the same machine. Obviously more sophisticated strategies can be adopted, e.g. the component can be deployed into different machines, in such case the method `createComponent` is replaced by the `getComponent` method.

**JSCL environment** We have presented above the primitives provided by `JSCL` for declaring reactive components and for coordinating them via event notification. Other systems have been introduced in [**?** ] to describe these issues (service declaration and coordination) in XML. On the one side, each component can be defined with an XML structure giving the signal types to which it reacts, the `iocl` support and the address to which it will be bound. These files are processed to obtain the corresponding Java skeleton code. On the other side, the coordination among components can be described in a separated XML document containing the definition of the services involved and their connections. Such file can be interpreted so to create the required coordination structure.

## 4 Long Running Transaction

`JSCL` has been adopted in [**?** ] for implementing a framework for Long Running Transactions. The deployment phase has been driven by Naïve Sagas [**?** ], a process calculus for compensable transactions, which defines Long Running Transactions in terms of logical blocks (*transactional flows*) orchestrating to reach a common goal. The building block of Naïve Sagas is the *compensation pair* construct. Given two actions $A$ and $B$, the compensation pair $A \div B$ corresponds to a process that uses $B$ as compensation for $A$. Intuitively, $A \div B$ yields two flows of execution: the *forward flow* and the *backward flow*. During the forward flow, $A \div B$ starts its execution by running $A$ and then, when $A$ finishes: $(i)$ $B$ is "installed" as compensation for $A$, and $(ii)$ the control is forwardly propagated to the other stages of the transactions. In case of a failure in the rest of the transaction, the backward flow starts so that the effects of executing $A$ must be rolled back. This is achieved by activating the installed compensation $B$ and afterward by propagating the rollback to the activities that were executed before $A$. Notice that $B$ is not installed if $A$ is not executed.

With `JSCL` the transactional blocks are obtained by suitable wrappers, `Transactional Gates` ($TG$), that use signal passing for activating the flows. The possible signal types that can be exchanged are: $sig_{FW}$, which activates the *forward flow*, $sig_{RB}$, for activating the *backward flow* (rollback), $sig_{CM}$, propagated to notify that the whole orchestration has been successful executed (commit) and $sig_{EX}$, exchanged to notify that the rollback phase has failed and the state of the whole transaction is inconsistent (also referred as abnormal termination). When a $TG$ receives a signal typed $sig_{FW}$, it tries to execute the main activity $A$; whenever the execution of $A$ normally terminates, the signal is propagated to the next stage. On the contrary, if $A$ throws an exception, a signal typed $sig_{RB}$ is propagated to the previous stage (the rollback is propagated in backward

way). Analogously when a $sig_{RB}$ is received by a $TG$, it tries to execute the *compensating activity* $B$, if it fails, throwing an exception, the signal is set to $sig_{EX}$ to notify that the rollback phase has failed and the state of the whole transaction is inconsistent, otherwise the rollback signal is propagated. The JSCL implementation of Naïve Sagas provides components implementing *parallel* and *sequential* structural composition of *transactional gates*. The composition constructs keep the structure of $TG$ and can be reused in further compositions. In Figure 6 it is shown how a Naïve Sagas compensable process $A_1 \div B_1; A_2 \div B_2; A_3 \div B_3$, is implemented using the JSCL graphical notation.

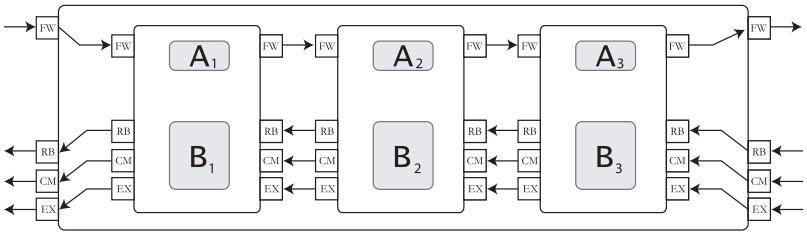


**Fig. 6.** JSCL Transactional Gates: an example.

The JSCL implementation of transactional flows can be formally described in SC as follows. We assume given two functions $[\![A]\!]_{fw}(x)$ and $[\![A]\!]_{rb}(x)$ that translate the Naïve Sagas atomic activity $A$ to SC internal behaviors, working on signal named $x$. We assume that the first function translates the successful return statements into the signal emission $\bar{x} : fw.0$ and the exception rising into $\bar{x} : rb.0$, and that the second function translates the successful return statements into the signal emission $\bar{x} : rb.0$ and the exception rising into $\bar{x} : ex.0$. The example Naïve Sagas compensable process, previously described, is so represented by the SC network $[\![P]\!]$:

$$[\![A_1 \div B_1]\!] \triangleq p_1[0]^{x:fw\to[\![A_1]\!]_{fw}(x)\ |\ x:rb\to[\![B_1]\!]_{rb}(x)}_{fw\triangleright p_2}$$

$$[\![A_2 \div B_2]\!] \triangleq p_2[0]^{x:fw\to[\![A_2]\!]_{fw}(x)\ |\ x:rb\to[\![B_2]\!]_{rb}(x)}_{fw\triangleright p_3 \bullet rb\triangleright p_1}$$

$$[\![A_3 \div B_3]\!] \triangleq p_3[0]^{x:fw\to[\![A_3]\!]_{fw}(x)\ |\ x:rb\to[\![B_3]\!]_{rb}(x)}_{rb\triangleright p_2}$$

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

## 5 Concluding remarks

We have introduced a core framework to formally describe coordination of distributed services. This framework has driven the implementation of a middleware for programming coordination policies by exploiting the event notification paradigm. In our approach the event notification paradigm supports fully distribution.

Unlike the current industrial technologies concerning service coordination (e.g. BPEL [**?** ]), our solution is based on top of a clear foundational approach. This should

provide strategies to prove coordination properties based on model checking or type systems. A semantic definition of the basic set of primitives can also drive the implementation of translators from industrial specification languages (e.g. WS-CDL [**?** ]) to our framework. Our approach differs from other event based proposals (e.g. SIENA [**?** ]), since focuses the implementation on the more distributed environment of services.
Our formal approach is based on process calculus literature (e.g. ccs [**?** ], $\pi$-calculus [**?** ]). Differently from $\pi$-calculus, the computation is boxed into components, in the style of Ambient Calculus [**?** ]. Moreover we avoid components nesting, to model a flat topology of the network and to provide a more loose-coupled programming model. As described before, SC models connections among components with peer-to-peer like structures (*flows*). Moreover, such flows can be logically grouped into multicast-channels, identified by the conversation schema. This provides a communication pattern closed to the one presented in [**?** ], even if SC does not deal with connection mobility.
There are several coordination models based on connection among components that provide dynamic reconfiguration (e.g. Reo [**?** ]). Our work mainly differs from Reo on the communication model adopted for composition. Reo is a channel based framework, while SC is an event based one. Hence, Reo handles component migration and channel management as basic notions, while SC focuses on the activities performed and on the coordination over dynamic network topologies.

In this paper, we focused on the coordination aspects. However the calculus SC can be enriched by considering signals as tagged nested lists [**? ?** ], which represent XML documents, and conversation schemata as abstractions for XML Schema types [**?** ]. This extension of conversation schemata lead to a more general notion of reaction based on pattern matching or unification in the style of [**?** ]. A further extension can provide component and schema name passing, modelling a more dynamic scenario.

SC can describe dynamic orchestrations trough reaction and flow update primitives. These primitives have effects only on the component view of the choreography, namely a component cannot update the reaction or the flow of another component. Flow management can be enriched providing a primitive to update remote flows. This primitive should spawn a *flow update envelope* into the network. The update of remote reaction is more difficult, since a reaction contains code and than is necessary to formalize and implement the code migration.