Programming services with correlation sets

Fabrizio Montesi and Marco Carbone

IT University of Copenhagen, Denmark
{fmontesi,carbonem}@itu.dk

Abstract. Correlation sets define a powerful mechanism for routing incoming communications to the correct running session within a server, by inspecting the content of the received messages. We present a language for programming services based on correlation sets taking into account key aspects of service-oriented systems, such as distribution, loose coupling, open-endedness and integration. Distinguishing features of our approach are the notion of correlation aliases and an asynchronous communication model. Our language is equipped with formal syntax, semantics, and a typing system for ensuring desirable properties of programs with respect to correlation sets. We provide an implementation as an extension of the JOLIE language and apply it to a nontrivial real-world example of a fully-functional distributed user authentication system.

1 Introduction

Correlation sets, introduced by WS-BPEL [17] (BPEL for short), are used to program routing policies for delivering incoming messages to the correct running session within a server. A message is relayed to an internal session whenever a part of its data content matches a part of the session state. These parts are defined by the correlation sets. Correlation sets are widely used in Service-Oriented Computing (SOC) and in web technologies, from complex multiparty interactions to simple client-server protocols between a web browser and a web server. Their role resembles that of unique keys in relational databases: they uniquely identify a session from a portion of their data. Considered in isolation there is little difference between the two concepts. The interesting aspect lies in the interplay with key aspects of SOC, such as distribution and loose coupling. The aim of this paper is to investigate this interplay, in order to gain insight on correlation sets as a programming methodology.

We develop a language for programming correlation-based services. Features of our approach are the direct manipulation of correlation data in programs and the notions of correlation aliasing. The former allows the programmer to write custom policies for instantiating correlation sets from within sessions, whereas the latter defines where correlation data is retrieved inside message content.

We start by analysing some prominent characteristics of SOC. The analysis is used as a foundation for reasoning on the basic constructs of our language and its semantics, whose structure takes inspiration from the π-calculus [14] and SOCK [7]. We establish a typing discipline that prevents the occurrence of some run-time errors, for example ensuring that a service does not break the property that each session is uniquely identifiable through a correlation set. Our results

show how to discipline message routing programming based solely on data for obtaining a determinism similar to that of π-calculus-like channels. We demonstrate applicability by providing an implementation of our language – in the form of an extension of the service-oriented language JOLIE [16] – and a nontrivial real-world example showing a fully-functional distributed user authentication system, inspired by the OpenID Authentication specifications [18].

1.1 Key concepts in Service-oriented Computing

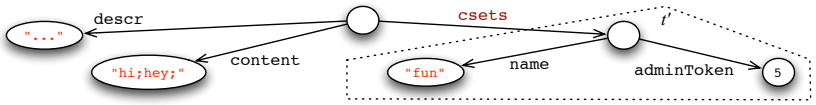
Distribution. Service-oriented architectures such as Web Services are in most cases distributed over a wide area network, e.g., the Internet. Distribution often features two main aspects: locality and communication asynchrony. The former means that processes run locally at different sites and use the network for communicating with each other. The latter means that messages are received at later points in time wrt when they were sent.
Sessions. Services may engage in multiple, concurrent interactions that may be complex and long-running, each one maintaining a separate execution state. Services support these interactions with *sessions*: stateful instances of workflows. Interactions between sessions are, in practice, dealt with correlation sets, which establish the session a message must be delivered to.
Loose Coupling. Services maintain minimal dependencies among each other, abstracting from internal implementation details. Particular emphasis is put on the types of the functionalities exposed by a service, each one defined by an *operation* and the *structure* of the data to be exchanged through it. The latter is usually defined as a tree using, e.g., the XML language.
Integration. SOC promotes reuse since new services are often implemented by composing already existing ones. Therefore, it is important to offer flexible mechanisms for adapting to the interfaces (e.g. APIs) of preexisting, legacy services and to recent technologies.
Open-Endedness. Service-oriented systems can be open-ended: participants may join or leave the system at run-time. For this reason, it is important to design languages and tools that allow safe execution of systems regardless of the environment they are executed in.

2 Language Overview

In this section, we outline with an example the main ideas of our language against the key concepts of SOC. Formal syntax and semantics will be given in § 3.

Our example is a common distributed scenario with a chat service supporting the management of chat rooms. Chat rooms are identified by name, as in IRC servers [1]. The service allows users to: create new chat rooms, publish a message in a chat room, retrieve published messages from existing chat rooms, and close chat rooms. When a client requests the creation of a chat room, the service checks that no other room with the same name exists. It then sends an *administration token* back to the invoker. Any client can publish messages in an open chat room or retrieve the history of published messages. The initial creator can close the chat room at any point by using the administration token.

Data Structures. Each chat room has a data structure representing its local state where its name, description, published messages, and administration token are stored. In our language, we represent data as trees where nodes are values of basic data types such as strings and integers. For instance, the state of a chat room is represented by the tree:



(1)

The root has three children pointed to by labels `descr`, `content` and `csets`. Subnode `csets` has two other children, `name` and `adminToken`. Data trees are accessed in programs by means of *paths*. Paths are sequences of edge names separated by dots, and can be used for traversing a tree starting from its root. Paths can be used in assignments and expressions. For example, the tree above could be initialised in our language with the following assignments:

```
descr = "..."; content = "hi;hey;";
csets.name = "fun"; csets.adminToken = 5
```

For brevity, we refer to a path as a variable, and the node it points to as its value. So in this case variable `content` would have value `"hi;hey;"`.

Communication Behaviour. In our language, data is exchanged between services by means of message passing. As in Web Services, messages are labelled by *operations*. Given operations `create`, `publish`, `read` and `close`, we could program the chat service behaviour as:

```
create(name)(csets.adminToken) { csets.adminToken = new };
run = 1; while( run ) {
    [publish(msg)] { content = content + msg.content + ";" }
    [read(req)(content) { 0 }] { 0 }
    [close(req)] { run = 0 }
}
```

The first instruction is an input on operation `create`. The content of the received message (a data tree) will be stored as a subtree of `name` which is a path in the local state. We call this input instruction a session start since its execution will start a new chat. Moreover, it is also a Request-Response (as in WSDL [20]): the client will wait for the server to reply with the content of `csets.adminToken` that is sent back once the local code in curly brackets `{ csets.adminToken = new }` is executed. `new` is a primitive that returns a locally-fresh token. After invocation, the service enters a loop containing a choice of three inputs with operations `publish` (for publishing in the chat room), `read` (for reading already published messages), and `close` (for closing the chat room). The inputs with operations `publish` and `close` are standard inputs called One-Way while the one with operation `read` is a Request-Response.

Dually to the server, we can give a sample code for a client:

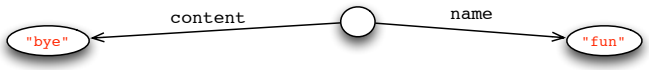
```
roomName = "MyRoom"; create@Chat(roomName)(adminToken);
msg1.roomName = roomName; msg1.content = "hi";
msg2.roomName = roomName; msg2.content = "hey";
```

```
{ publish@Chat(msg1) | publish@Chat(msg2) };
read@Chat(roomName)(chatContent); close@Chat(adminToken)
```

This client sample performs a Solicit-Response output (dual of Request-Response) on operation `create`. The message is sent at location `Chat`, the location of the chat server. Locations (cf. URIs) define where services are deployed, modeling locality. The instruction is completed when the response from the server is received and assigned to `adminToken`. Thereafter, the client sends messages to the chat with two Notification outputs (dual to One-Way) executed in parallel by means of the `|` operator. Finally, the client reads the content of the chat room through operation `read` and closes it by means of operation `close`.

In our language, messages are delivered asynchronously to sessions. After a message is sent, it is guaranteed that the receiving service has buffered it, but not that a session has consumed it. This can lead to bad behaviour. For this reason, the semantics of our language in § 3 preserves ordering of buffered messages.

Correlation Sets and Aliasing. The chat service may have many running sessions executing in parallel, each one representing a chat room. How can it identify the session an incoming message is for, when it receives one from the network? Correlation sets address this issue. In our language, a correlation set is a set of paths, called correlation variables, that define which nodes of a session state identify the session. A correlation set is defined by means of the keyword `cset`. Our chat service has two correlation sets: `cset {name}` and `cset {adminToken}`. For example, if the chat server receives a message carrying the tree:



the first correlation set will then associate the message to the session running with the state shown in (1), since both message and session share the same value for correlation variable `name`, and route the message to it. We call this association *correlation*, and we say that the message *correlates* with the session. The value for correlation variable `name` is stored in the subtree `csets` in the session state. More generally, in our language every correlation value must be put in that subtree. This makes modifications to data that influences correlation explicit. We exploit this aspect in the definition of our type system, in § 4.

Correlation sets are specified by the receiver: the client does not need to be aware of the correlation sets of the invoked service but needs only to send messages with the expected data structures, enabling loose coupling. Correlation sets are also prone to integrate with existing technologies. For example, a web server session can be identified by the correlation set $c = \{\texttt{sid}\}$, the session id usually stored in a browser cookie.

Above, we associated a message to a session by matching the value of the same path `name` in the message tree and the session state. Such a mechanism is limiting, because the fact that the two paths must be the same means that there is tight coupling between the service implementation and its interface. This could be even completely unfeasible. Consider, for instance, the case in which a programmer must write a service that interacts with a legacy application. The interface of the service will have to be in accordance to what the legacy application expects. Let us assume now that the legacy application will send two different kinds of message to our new chat service, on different operations.

The first contains the room name under path `roomName` and the other in the root of the message data tree; this is the behaviour of the client that we showed before. How can we relate both values to the same correlation variable? We address this issue with a notion of aliasing: a correlation variable may be defined together with a list of aliases that tell where to retrieve, in a message, the value to be compared with that of the session, depending on the type of the incoming message (aliasing can be looked at as a type itself). Hence, the correlation set definitions for the chat service become:

```
cset { name:Create Publish.roomName Read }    cset { adminToken:Close }
```

where, for brevity, we assume the input message type of each operation has the same name with an uppercase initial. Data types will be presented in detail in § 5. Correlation aliasing is a key feature for meeting the requirements of integration.

3 Data Structures, Syntax and Semantics

In this section, we formalise data, syntax and give the semantics of our language.

Data Trees and Correlation. Let t range over a set of *data trees* $\mathcal{T}$, with edges denoted by $\mathtt{x}$,$\mathtt{y}$,$\mathtt{z}$,... and nodes denoted by v. v is a value, which can be a string, an integer, a *location* or the undefined value $v_\perp$. Values is the set of all values. In programs, data trees are accessed by *paths*. A path $\mathtt{p}$ is a sequence of tree edges $\mathtt{x}_1.\ \ldots\ .\mathtt{x}_n$ denoting an endofunction on data trees defined as:

$$\mathtt{p}(t) = \begin{cases} t & \text{if } \mathtt{p} = \epsilon \\ \mathtt{p}'(t') & \text{if } \mathtt{p} = \mathtt{x}.\mathtt{p}' \text{ and } \mathtt{x} \text{ is an edge from the root of } t \text{ to } t\text{'s subtree } t' \\ t_\perp & \text{if } \mathtt{p} = \mathtt{x}.\mathtt{p}' \text{ and there is no edge } \mathtt{x} \text{ from } t \text{ to a subtree } t' \end{cases}$$

where ϵ denotes the empty sequence and $t_\perp$ a tree with a single node with value $v_\perp$. We denote the set of possible paths with Paths. Furthermore, we require paths written in programs to be nonempty. We extract the value of the root of a tree by using the function $^\dagger : \mathcal{T} \to \mathsf{Values}$.

A *correlation set* c is a set of paths corresponding to those values that identify a running session of a service: $c \subseteq \mathsf{Paths}$. A service may define more than one correlation set: we denote with C a set of correlation sets, $C \subseteq \mathcal{P}(\mathsf{Paths})$.

We model correlation aliasing by means of an *aliasing function*, α_C, that establishes where to retrieve correlation values in a message received for an operation. Let $\mathcal{O}$ be the set of possible operations, ranged over by $\mathtt{o}$. An aliasing α_C is a function that given an operation $\mathtt{o}$ returns a correlation set $c \in C$ and a function from paths contained in c to paths in the incoming message:

$$\alpha_C : \mathcal{O} \rightharpoonup C \times (\mathsf{Paths} \rightharpoonup \mathsf{Paths})$$

The aliasing function α_C bases aliases on operations, and not on message types like in § 2. This is a matter of convenience: in our language implementation, aliases are defined on message types and are converted exactly to an aliasing function as described in this section.

We now present our definition of correlation in terms of the relation $\vdash_{\alpha_C}$:

Definition 1 (Correlation $\vdash$). *A data tree t' received for operation* `o` correlates *with a data tree t with respect to an aliasing α_C, written $t', \mathtt{o} \vdash_{\alpha_C} t$, whenever*

$$\exists c, f.\ c \neq \emptyset \ \wedge\ \alpha_C(\mathtt{o}) = (c, f) \ \wedge\ \forall \mathtt{p} \in c.\ f(\mathtt{p})(t')^\dagger = \mathtt{csets.p}(t)^\dagger \neq v_\perp$$

Syntax. The syntax of programs is structured in three layers. The *behavioural layer* models actions performed by a service session, the *service layer* handles the definition of correlation sets, state and session instantiation and the *network layer* deals with deployment and communication. This layering is reflected in the language implementation, presented in § 5.

Behavioural layer. Behavioural terms are given as processes, ranged over by $P, Q, \ldots$ and defined by the grammar below where r denotes a *channel name*, $l, l', \ldots$ *locations* and $e, e', \ldots$ unspecified first-order expressions that include locations and an operator `new` for generating locally-fresh values:

$P, Q, \ldots ::=$	$\sum_i \mathtt{[}\eta_i\mathtt{]}\{P_i\}$	(choice)	$\eta ::=$ `o(p)`	(one-way)
			$\mid$ `o(p)(p')` $\{P\}$	(request-response)
$\mid$	η	(input)		
$\mid$	$\overline{\eta}$	(output)	$\overline{\eta} ::=$ `o@p(p')`	(notification)
$\mid$	`if(`e`)` $\{P\}$ `else` $\{Q\}$	(cond)	$\mid$ `o@p(p')(p'')`	(solicit-response)
$\mid$	`while(`e`)` $\{P\}$	(loop)		
$\mid$	`p=`e	(assign)	$e ::=$ `new`	(new)
$\mid$	$P; Q$	(seq)	$\mid$ l	(location)
$\mid$	$P \mid Q$	(par)	$\mid$ $\ldots$	(first-order expr)
$\mid$	$\mathbf{0}$	(inact)		

Input-guarded branching is available through (choice). Communications can be unidirectional (one-way) or bidirectional (request-response). `o(p)` reads an incoming message for operation `o` and places the received tree in the local state tree under path `p`. Dually, `o@p(p')` sends a message for operation `o` to the location stored in the state node `p` points to, carrying the data in the local state pointed by `p'`. Alternatively, (request-response) and (solicit-response) allow for Request-Response communications. All other constructs are standard.

Service layer. Services, denoted by S, consist of a service behaviour definition (replicated process) and an aliasing α_C, defining correlation sets and aliasings:

$$S ::= \sum_i \mathtt{[}\eta_i\mathtt{]}\{P_i\} \triangleright_{\alpha_C} \mathbf{0} \quad \text{(service)} \qquad \mid \qquad \mathbf{0} \triangleright_{\alpha_C} P \cdot t_\perp \cdot \epsilon \quad \text{(starter)}$$

Normally, services become active only after they are invoked. For this reason, a system needs at least one service to spontaneously start invoking other services. We call such a service a *starter*. A starter specifies a single session, which will start its execution without the need to be triggered.

Network layer. Services are *deployed* on locations and composed in parallel to form networks:

$$N, M ::= [S]_l \quad \mid \quad \boldsymbol{\nu} r\ N \quad \mid \quad N \mid N \quad \mid \quad \mathbf{0} \quad \text{(network)}$$

We assume that, for any network, it is never the case that two services are deployed with the same location.

Semantics. We extend the language syntax with run-time terms (as in [8]):

$$\begin{array}{lll} S ::= P \triangleright_{\alpha_C} I & & \text{(running service)} \\ I ::= P \cdot t \cdot \tilde{m} \quad | \quad I \mid I & & \text{(running sessions)} \\ P ::= \ldots \mid \mathsf{Wait}(r, \mathtt{p}) \mid \mathsf{Exec}(r, \mathtt{p}, P) & & \text{(running processes)} \end{array}$$

Services are extended to support multiple locally running sessions (denoted by I). Each session consists of the currently executing run-time process, a state t and a FIFO queue $\tilde{m}$ [8], with ϵ representing the empty queue. m is a message of the form $(r, \mathtt{o}, t)$ where r is a channel, $\mathtt{o}$ an operation and t the content. The terms $\mathsf{Wait}(r, \mathtt{p})$ and $\mathsf{Exec}(r, \mathtt{p}, P)$ model Request-Response communications.

We equip our model with a structural congruence defined as the least congruence relation on P, I and N such that $(\mid, \mathbf{0})$ is a commutative monoid, it supports alpha-conversion, $\mathbf{0}; P \equiv P$, $P \equiv P'$ and $I \equiv I'$ imply $[P \triangleright_{\alpha_C} I]_l \equiv [P' \triangleright_{\alpha_C} I']_l$, $\boldsymbol{\nu} r \, \boldsymbol{\nu} r' N \equiv \boldsymbol{\nu} r' \boldsymbol{\nu} r \, N$ and such that $(\boldsymbol{\nu} r \, N) \mid N' \equiv \boldsymbol{\nu} r \, (N \mid N')$ if $r \notin \mathtt{cn}(N')$, where $\mathtt{cn}$ is a function that returns the set of channel names in a term.

We give the semantics in terms of a labeled transition system (lts). The labels, ranged over by μ, are standard and their domain is omitted. The *behavioural layer* defines the semantics of service sessions. A selection of the rules is reported below (all rules can be found in the online appendix [2]).

$$\text{(P-Choice)} \;\; j \in J \quad \eta_j \xrightarrow{\mu} Q_j \quad \Rightarrow \quad \textstyle\sum_{i \in J} [\eta_i]\{P_i\} \xrightarrow{\mu} Q_j; P_j$$

$$\text{(P-Solicit)} \;\; \mathtt{o@p(p')(p'')} \xrightarrow{\boldsymbol{\nu} r \; \mathtt{o@p(p')}} \mathsf{Wait}(r, \mathtt{p''}) \qquad \text{(P-Notify)} \;\; \mathtt{o@p(p')} \xrightarrow{\boldsymbol{\nu} r \; \mathtt{o@p(p')}} \mathbf{0}$$

$$\text{(P-Req)} \;\; \mathtt{o(p)(p')}\,\{P\} \xrightarrow{r:\mathtt{o(p)}} \mathsf{Exec}(r, \mathtt{p'}, P) \qquad \text{(P-OneWay)} \;\; \mathtt{o(p)} \xrightarrow{r:\mathtt{o(p)}} \mathbf{0}$$

$$\text{(P-EndExec)} \;\; \mathsf{Exec}(r, \mathtt{p}, \mathbf{0}) \xrightarrow{\overline{r}\,\mathtt{p}} \mathbf{0} \qquad \text{(P-Wait)} \;\; \mathsf{Wait}(r, \mathtt{p}) \xrightarrow{r\,\mathtt{p}} \mathbf{0}$$

$$\text{(P-Exec)} \;\; P \xrightarrow{\mu} P' \;\Rightarrow\; \mathsf{Exec}(r, \mathtt{p}, P) \xrightarrow{\mu} \mathsf{Exec}(r, \mathtt{p}, P') \qquad \text{(P-Asgn)} \;\; \mathtt{p} = e \xrightarrow{\mathtt{p}=e} \mathbf{0}$$

Rules `P-OneWay` and `P-Notify` allow, respectively, for the receiving and sending of asynchronous one-way messages. Rules `P-Req` and `P-Solicit` do similarly for Request-Response patterns, handling also the subsequent response computation and sending. The computation of the response is handled by rule `P-Exec`; when the response computation terminates, the caller and the callee communicate again by means of the private channel that they established in their interaction. The modeling of Request-Response replies through private channels supports classic client-server communications, where the client could be unable to expose inputs of its own due to external restrictions, e.g. firewalls.

The *service layer* interfaces a session behaviour with the hosting service. Below, we give a selection of the rules (complete table in the appendix [2]):

$$\text{(S-Get)} \;\; P \xrightarrow{r:\mathtt{o(p)}} P' \;\Rightarrow\; P \cdot t \cdot (r, \mathtt{o}, t') :: \tilde{m} \xrightarrow{\tau} P' \cdot t \leftarrow_{\mathtt{p}} t' \cdot \tilde{m}$$

$$\text{(S-Send)} \;\; P \xrightarrow{\boldsymbol{\nu} r \; \mathtt{o@p(p')}} P' \;\Rightarrow\; P \cdot t \cdot \tilde{m} \xrightarrow{\boldsymbol{\nu} r \; \mathtt{o@p}(t)^{\dagger}(\mathtt{p'}(t))} P' \cdot t \cdot \tilde{m}$$

$$\text{(S-SR)} \;\; P \xrightarrow{\overline{r}\,\mathtt{p}} P' \;\Rightarrow\; P \cdot t \cdot \tilde{m} \xrightarrow{\overline{r}\,\mathtt{p}(t)} P' \cdot t \cdot \tilde{m}$$

$$\text{(S-RR)} \;\; P \xrightarrow{r\,\mathtt{p}} P' \;\Rightarrow\; P \cdot t \cdot \tilde{m} \xrightarrow{r\,t'} P' \cdot t \leftarrow_{\mathtt{p}} t' \cdot \tilde{m}$$

$$\text{(S-Asgn)} \;\; P \xrightarrow{\mathtt{p}=e} P' \;\Rightarrow\; P \cdot t \cdot \tilde{m} \xrightarrow{\tau} P' \cdot t \twoheadleftarrow_{\mathtt{p}} e(t) \cdot \tilde{m}$$

$$\text{(S-Corr)} \;\; t', \mathtt{o} \vdash_{\alpha_C} t \;\Rightarrow P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} \xrightarrow{\boldsymbol{\nu} r \; \mathtt{o}(t')} P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} :: (r, \mathtt{o}, t')$$

$$(\texttt{S-Start})\quad \frac{t,\mathtt{o} \not\vdash_{\alpha_C} I \qquad P \xrightarrow{r:\mathtt{o}(\mathtt{p})} P' \qquad t' = \mathsf{init}(t,\mathtt{o},\alpha_C)}{P \triangleright_{\alpha_C} I \xrightarrow{\boldsymbol{\nu} r\ \mathtt{o}(t)} P \triangleright_{\alpha_C} I \mid P' \cdot t_\perp \leftarrow_{\mathtt{p}} t \leftarrow_{\mathtt{csets}} t' \cdot \epsilon}$$

$$\mathsf{init}(t,\mathtt{o},\alpha_C) = \begin{cases} t_\perp \twoheadleftarrow_{\mathtt{p}_1} f(\mathtt{p}_1)(t) \ldots \twoheadleftarrow_{\mathtt{p}_n} f(\mathtt{p}_n)(t) & \text{if } \alpha_C(\mathtt{o}) = (\{\mathtt{p}_1,\ldots,\mathtt{p}_n\}, f) \\ t_\perp & \text{if } \mathtt{o} \notin \mathit{Dom}(\alpha_C) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In all rules but `S-Corr` and `S-Start` we have omitted the whole service structure (it is irrelevant for those rules). Rule `S-Start` implements the spawning of a new local session by receiving a message that does not correlate with any running session (thus giving precedence to existing sessions), initialising its `csets` subtree if there is an aliasing definition for operation `o`. Note that the initialisation function $\mathsf{init}(t,\mathtt{o},\alpha_C)$ is partial and undefined if the message does not contain all the correlation data specified in α_C for `o`; in this case, rule `S-Start` can not be applied. The relation $t',\mathtt{o} \not\vdash_{\alpha_C} I$ is defined whenever there is no state t in I such that $t',\mathtt{o} \vdash_{\alpha_C} t$. Moreover, $t \leftarrow_{\mathtt{p}} t'$ is a function that returns a new tree obtained from t by replacing the subtree pointed by `p` with t'; the function automatically creates the missing nodes for traversing t with `p`, initializing them with $v_\perp$. Function $t \twoheadleftarrow_{\mathtt{p}} e(t')$ does the same but replaces only $\mathtt{p}(t)$'s root with the value that results from the evaluation of e on t', $e(t')$. Rule `S-Get` allows a running process to fetch the first element from the message queue of its session. Rule `S-Send` propagates the label for a sending, which will be used by the network layer for performing the actual message transmission; the rule substitutes the paths `p` and `p'` in the original label with, respectively, the location pointed by `p` and the data tree pointed by `p'` stored in the session state. Rules `S-Send-Resp` and `S-Recv-Resp` close a Request-Response communication by exchanging the final reply. Rule `S-Asgn` models variable assignment. Rule `S-Corr` allows a running session to receive a correlating message and store it in its local queue (we omit the condition for handling the special case of an empty queue $\tilde{m} = \epsilon$).

The outer layer of our semantics, the *network layer*, deals with inter-service interactions. The rules are standard and can be found in the appendix [2].

4 Properties and Types

In this section we discuss some desirable properties of services that can be captured with our language. Some of them are based on conditions that need to be guaranteed through the use of a typing system.

Properties. Our properties focus on integrity of sessions and communications.

Property 1 (Message delivery atomicity). Let $N \equiv \boldsymbol{\nu}\tilde{r}\ ([S_1]_{l_1} \mid M)$ *such that* $S_1 \xrightarrow{\boldsymbol{\nu} r'\ \mathtt{o}@l_2(t_M)} S_1'$ *and* $N \xrightarrow{\tau} \boldsymbol{\nu}\tilde{r}\,\boldsymbol{\nu} r'\ ([S_1']_{l_1} \mid M')$. *Then,* $M \equiv [S_2]_{l_2} \mid M''$, $M' \equiv [S_2']_{l_2} \mid M''$ *and either (i)* $S_2 \equiv P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} \wedge S_2' \equiv P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} :: (r,\mathtt{o},t_M)$ *or (ii)* $S_2 \equiv P \triangleright_{\alpha_C} I \ \wedge\ S_2' \equiv P \triangleright_{\alpha_C} I \mid P' \cdot t \leftarrow_{\mathtt{p}} t_M \cdot \epsilon$ *for some* $t, \mathtt{p}$.

Property 1 states that if a service successfully executes a message sending then there is another service in the network that either (i) put the message in the queue of a correlating session or (ii) started a new session with a state containing the message data. This is guaranteed by our semantics since a message sending is completed only by synchronising with the receiver by means of rule `S-Start` or rule `S-Corr`.

Property 2 (No session ambiguity). For each t', $\mathtt{o}$ and service $P \triangleright_{\alpha_C} I$, there is at most one running session $P' \cdot t \cdot \tilde{m}$ in I such that $t', \mathtt{o} \vdash_{\alpha_C} t$.

Our second property states that a service can never have more than one running session that correlates with the same message. Such a situation would lead to non-deterministic assignment of incoming messages, which goes against the principle that a session is *uniquely* identifiable under correlation.

Property 3 (Possible inputs). Let $S \equiv P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m}$. If $P' \xrightarrow{r:o(p)} P''$ then $\tilde{m} = \tilde{m}' :: (r, \mathtt{o}, t') :: \tilde{m}''$ $\vee$ $S \xrightarrow{\nu r\, o(t')} P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} :: (r, \mathtt{o}, t')$.

Property 3 says that if a session needs to perform an input, then a message for that input is in a queue and/or the enclosing service is able to receive a message for the session by correlation. I.e., whenever a session tries to perform an input its state has the related correlation set fully instantiated.

Properties 2 and 3 depend on the states of the sessions running in a service. Bad programming can lead to executions for which the properties do not hold. For example, for $\alpha_C = [\mathtt{join} \mapsto (\{\mathtt{x}\}, [\mathtt{x} \mapsto \epsilon])]$, if the service with behaviour `start(a);csets.x = 5;join(b)` gets invoked twice on `start`, it will spawn two sessions which will both execute `csets.x = 5`. After that, by α_C, both sessions can correlate with a message for operation `join` with value `5` as root node. This situation breaks property 2 and leads to the non-deterministic routing. Also, if $\alpha_C = [\mathtt{join} \mapsto (\{\mathtt{x}, \mathtt{y}\}, [\mathtt{x} \mapsto \epsilon, \mathtt{y} \mapsto \mathtt{y}])]$, we break property 3: the two sessions would be stuck forever waiting for a message for `join`, because rule `S-Corr` could never be applied due to the lack of a value for `y` in the sessions.

Typing System. We present a type system that focuses on the manipulation of correlation data. Our typing performs an initialisation analysis for correlation variables. Although this is a well-established technique, our setting requires particular attention to the concurrent execution of multiple sessions, and the interplay between session behaviour and the aliasing function.

Typing judgments have the form $\Gamma \vdash P : \Delta_N | \Delta_P$, where $\Delta_N \subseteq \mathsf{Paths}$ and $\Delta_P \subseteq \mathsf{Paths} \times \{\circ, \bullet\}$. Δ_N says which correlation paths *need* to be initialised before P executes and Δ_P contains the correlation paths initialised (*provided*) by P. In Δ_P each correlation path is flagged telling if it carries a fresh value ($\circ$) or not ($\bullet$). The main typing rules follow (all rules and an extended discussion can be found in the appendix [2]).

$$\text{(T-CSets-New)}\ \frac{}{\Gamma \vdash \mathtt{csets.p{=}new} : \emptyset | \{\mathtt{p}^\circ\}} \qquad \text{(T-CSets-Expr)}\ \frac{e \text{ not undefined}}{\Gamma \vdash \mathtt{csets.p}{=}e : \emptyset | \{\mathtt{p}^\bullet\}}$$

$$\text{(T-Seq)}\ \frac{\Gamma \vdash P : \Delta_{N_1} | \Delta_{P_1} \quad \Gamma \vdash Q : \Delta_{N_2} | \Delta_{P_2} \quad \Delta' = (\Delta_{N_2} \setminus \Delta_{P_1}) \cup \Delta_{N_1}}{\Gamma \vdash P;Q : \Delta' | \Delta_{P_1} \uplus \Delta_{P_2}}$$

$$\text{(T-OneWay)}\ \frac{\Gamma(\mathtt{o}) = c \neq \emptyset \quad \mathtt{p} \neq \mathtt{csets.p}'}{\Gamma \vdash \mathtt{o(p)} : c | \emptyset} \qquad \text{(T-OneWay-Start)}\ \frac{(\Gamma \vdash \mathtt{o(p)} : c | \emptyset \ \vee\ c = \emptyset) \ \wedge\ \mathtt{p} \neq \mathtt{csets.p}'}{\Gamma \vdash_s \mathtt{o(p)} : c | \emptyset}$$

$$\text{(T-Service)}\ \frac{\forall j \in J.\ \begin{cases} \Gamma_j \vdash_s \eta_j : c_j | \Delta_{P_j} \ \wedge\ \Gamma_j \vdash P_j : \Delta'_{N_j} | \Delta'_{P_j} \ \wedge\ \Delta'_{N_j} \subseteq c_j \uplus \Delta_{P_j} \ \wedge \\ \forall \mathtt{o} \in Dom(\Gamma_j).\, \alpha_C(\mathtt{o}) = (\Gamma_j(\mathtt{o}), f) \ \wedge\ Dom(f) = \Gamma_j(\mathtt{o}) \ \wedge\ \Delta_{P_j} \uplus \Delta'_{P_j} \ltimes C \ \wedge \\ \forall i \in J.\ i \neq j \Rightarrow \Gamma_i \vdash_s \eta_i : c_i | \Delta_{P_i} \ \wedge\ \Gamma_i \vdash P_i : \Delta'_{N_i} | \Delta'_{P_i} \ \wedge\ (\Delta_{P_i} \cup \Delta_{P_j}) \cap c_j = \emptyset \end{cases}}{\emptyset \vdash \sum_{i \in J} [\eta_i]\{P_i\} \triangleright_{\alpha_C} I : \emptyset | \emptyset}$$

The first two rules check the freshness of a correlation variable initialisation. In `T-CSets-Expr` we require e to be defined, i.e. that its evaluation will not yield $v_\perp$. This is a simple (but omitted) definite assignments analysis. Rule `T-Seq` checks that a same correlation variable is not defined multiple times – $\Delta_{P_1} \uplus \Delta_{P_2}$ – and

propagates the set of variables that need initialisation before executing $P;Q$. The disjoint union operator $\uplus$ behaves as the union operator $\cup$, but is defined only if the two sets are disjoint. The operator ignores the freshness flag; as such, it is never the case that a same path $\mathtt{p}$ can be in $\Delta_P \uplus \Delta'_P$ more than once (either with the same flag or with two different flags). We assume the same behaviour for the other sets operators (union, intersection and subtraction). Rule `T-OneWay` uses the environment Γ to register a requirement for the aliasing function of the enclosing service, i.e. that a nonempty correlation set c is associated to operation $\mathtt{o}$. It sets $\Delta_N = c$ for the input statement, meaning that the latter requires all the nodes pointed by the paths in c to be initialised in the state before being executed. These requirement are relaxed if the operation is used as a guard for starting a new session, in rule `T-OneWay-Start`, for allowing starting operations to have an empty associated correlation set. Rule `T-Service` checks that a service is well-typed. First, it checks that every branch is well-typed and that the aliasing function α_C complies to the requirements stored in the environment Γ_j of each branch. Then, it checks that for each correlation set $c \in C$ that will be completely defined at least one path in c will point to a fresh value in the session state, ensuring that sessions will be distinguishable by correlation. This check is performed through $\Delta_P \ltimes C$; relation $\ltimes$ is formally defined below. Finally, the rule forbids different initialisation methods of a same correlation set, i.e. when two different session branches use the same correlation set they must agree if that correlation set is initialised through local assignments or through the message that started the session.

Relation $\ltimes$ captures that, for the sake of being uniquely identifiable by correlation, a session needs at least one correlation variable to be fresh for every correlation set that is completely initialised.

Definition 2 (Correlation set freshness relation $\ltimes$).

$$\Delta_P \ltimes C \quad \textit{iff} \quad \forall c \in C \,.\, (\nexists \mathtt{p} \in c \,.\, \mathtt{p} \notin \Delta_P) \;\Rightarrow\; (\exists \mathtt{p} \in c \,.\, \mathtt{p}^\circ \in \Delta_P)$$

We introduce now a notion of error in the semantics adding two rules. Both use a `wrong` label that carries the location of the originating service. The first rule requires that a service has two running sessions that correlate with the same message t_M for the same operation $\mathtt{o}$, the negation of property 2. The second rule, instead, is active when a running session wishes to input on an operation for which there is no message in the queue and the correlation mechanism can route no new message to such a session, the negation of property 3.

$$(\texttt{S-Wrong-Corr}) \quad \frac{P \triangleright_{\alpha_C} I \mid P' \cdot t' \cdot \tilde{m}' \xrightarrow{\nu r\, \mathtt{o}(t_M)} P \triangleright_{\alpha_C} I \mid P' \cdot t' \cdot \tilde{m}' : (r, \mathtt{o}, t_M) \qquad P \triangleright_{\alpha_C} I \mid P'' \cdot t'' \cdot \tilde{m}'' \xrightarrow{\nu r'\, \mathtt{o}(t_M)} P \triangleright_{\alpha_C} I \mid P'' \cdot t'' \cdot \tilde{m}'' : (r', \mathtt{o}, t_M)}{[P \triangleright_{\alpha_C} I \mid P' \cdot t' \cdot \tilde{m}' \mid P'' \cdot t'' \cdot \tilde{m}'']_l \xrightarrow{\text{wrong } l} [P \triangleright_{\alpha_C} I \mid P' \cdot t' \cdot \tilde{m}' \mid P'' \cdot t'' \cdot \tilde{m}'']_l}$$

$$(\texttt{S-Wrong-Input}) \quad \frac{P' \xrightarrow{r:\mathtt{o}(p)} P'' \;\; \tilde{m} \neq \tilde{m}' : (r, \mathtt{o}, t') : \tilde{m}'' \;\; P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} \xrightarrow{\nu r \, \mathtt{o}(t'')} P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m} : (r, \mathtt{o}, t'')}{[P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m}]_l \xrightarrow{\text{wrong } l} [P \triangleright_{\alpha_C} I \mid P' \cdot t \cdot \tilde{m}]_l}$$

We can finally show the main results of our type system:

Theorem 1 (Subject Reduction).
$\Gamma \vdash N : \Delta_N | \Delta_P \;\wedge\; N \xrightarrow{\mu} N' \;\Rightarrow\; \Gamma \vdash N' : \Delta'_N | \Delta'_P$.

Theorem 2 (Safety). *Let l be a location in N.*

1. $\Gamma \vdash N : \Delta_N | \Delta_P \;\Rightarrow\; N \overset{\texttt{wrong}\,l}{\not\rightarrow} N'$
2. $\Gamma \vdash N : \Delta_N | \Delta_P \;\Rightarrow\; N | N' \overset{\texttt{wrong}\,l}{\not\rightarrow} N''$

Note that Theorem 2.2 ensures *local safety*, i.e. that a well-typed network will respect our properties regardless of its context.

5 Language Implementation in JOLIE

We have implemented the techniques described in the previous sections by extending the JOLIE programming language. Our solution is now the official mechanism for programming correlation with JOLIE.

The JOLIE Language. We give a brief description of JOLIE [16, 11], an open source [5] fully-fledged service-oriented programming language.

JOLIE programs are composed by a *behavioural part* and a *deployment part*. The syntax of the behavioural part is a superset of the syntax of our behavioural layer. When it comes to correlation, the only relevant difference is that output primitives refer to output ports (described below) rather than using paths for pointing to locations. The deployment part defines *communication ports* and *interfaces*. *Output ports* are used for invoking external services, whereas *input ports* are used to expose locations on which the service can receive messages. A port specifies a location and the data protocol to use (e.g. SOAP [19]). Some JOLIE protocol implementations allow for configuring protocol-specific headers. For example, we can connect cookie values in HTTP to paths in message data trees. As a consequence, programmers can store correlation values as cookies in web browsers that invoke a JOLIE service, thus seamlessly integrating our approach with common web technologies. Interfaces describe the (types of the) operations used in the behavioural part. Each element in an interface couples an operation to its message types, which are structured as trees. For example, the following defines an interface with a Request-Response operation `sum` that takes a tree with two subnodes – `x` and `y` – and returns an integer:

```
type SumRequest:void { .x:int .y:int }
interface SumInterface {RequestResponse:sum(SumRequest)(int)}
```

Correlation Set Definitions. We have implemented the syntax exposed in § 2 for defining correlation sets based on message types. A `cset` block corresponds to the definition of a correlation set c in our model and its related aliases in α_C:

$$\texttt{cset}\ \{\ \textit{list of}\ V\ \}\qquad V ::= \texttt{p} : \textit{list of}\ \texttt{p}_T$$

where `p` is a path and $\texttt{p}_T$ a path that starts with a message type reference. In our implementation, these definitions are converted to an aliasing function α_C for the interpreter; this is a convenient shortcut: if two operations share a same message type, then the aliasing function generated for the interpreter will have an entry for both operations with the same aliasing specified for that type by the programmer. Furthermore, we implemented a check for verifying that an alias is compatible with the related message type – i.e. if the message type contains the node of interest pointed by $\texttt{p}_T$. Thus we can ensure that a correlation set definition is compatible with the interface provided by the service.

Primitive `new` has been implemented using the Java standard library for generating secure Universally Unique Identifiers (UUID).

Implementation. The JOLIE interpreter is developed in the Java language and is structured in four modules. The *Parsing* module reads a program, produces its related AST (Abstract Syntax Tree), analyzes it and generates an OOIT. An OOIT (*Object-Oriented Interpretation Tree*) executes a session behaviour. The *Runtime Environment* handles the creation of sessions and their execution states. The *Communication Core* handles communications.

We updated the Parsing module for handling the syntax for defining correlation sets, applying our type system and converting `cset` definitions into an aliasing function α_C for the Runtime Environment. The Java class used by the Runtime Environment for controlling the execution of a session has been augmented with a message queue. When a node from the OOIT asks for a message input, the Runtime Environment checks the message queue of its session as specified by rule `S-Get`. Message queues are filled with incoming messages received by the Communication Core, looking for correlation as defined in rule `S-Corr`. If no correlating session is found, the Runtime Environment is asked to start a new session with the message, cf. `S-Start`. If the session can not be started, a fault `CorrelationError` is sent to the invoker and the message is discarded[1].

Request-Response interactions are supported by means of communication channel objects. The OOIT nodes implementing rules `P-Request` and `P-Solicit` (and their continuation) are given access to the channel of interest and can use it for sending or receiving responses as specified by rules `P-End-Exec` and `P-Wait`, abstracting from the underlying details.

6 Example: a decentralised authentication protocol

We present now an example inspired by the OpenID Authentication specifications [18]. OpenID is a largely adopted Single Sign-On solution based upon a decentralised authentication protocol that allows a service, called *relying party*, to authenticate a user, the *client*, by relying on another external service that is responsible for handling identities, the *identity provider*. When the client requests access to the relying party, the latter opens an authentication session in the identity provider. The client can then send its authentication credentials to the session in the identity provider, which will inform the relying party on the result of the authentication attempt.

We implemented the protocol in the updated version of JOLIE. The example can be downloaded at [3], where we support web browser clients by means of the JOLIE integration with HTTP.

The code below is a sketch of the relying party service:

```
cset { clientToken: ... }
cset { secureToken: AuthMessage.secureToken }
interface RelyingPartyInterface {
OneWay: authSucceeded(AuthMessage), authFailed(AuthMessage)
RequestResponse: login(LoginRequest)(Redirection) }
main {
  login( loginRequest )( redirection ) {
```

[1] For space reasons, we do not report fault semantics in this paper. The implementation is in line with the fault handling semantics of JOLIE, reported in [15, 6].

```
    clientToken = new; secureToken = new;
    openRequest.relyingPartyIdentifier = MY_IDENTIFIER;
    openRequest.clientToken = csets.clientToken;
    openRequest.secureToken = csets.secureToken;
    openAuth@IdentityProvider( openRequest );
    /* ... build redirection message for client ... */
  }; [ authSucceeded( message ) ] { /* ... */ }
     [ authFailed( message ) ]    { /* ... */ }
}
```

First, the service receives a request on the Request-Response operation `login` from the client for initiating the protocol. The body of `login` generates two fresh tokens: `clientToken`, referred by the first correlation set[2], and `secureToken`, referred by the second one. We will use `clientToken` for receiving messages from the client and `secureToken` for receiving messages from the identity provider. The client is not informed about `secureToken`, preventing it to maliciously act as the identity provider. The body of `login` performs a call to the identity provider, opening an authentication session and communicating `secureToken`. We can now safely reply to the client that invoked operation `login`: property 1, from § 4, guarantees that the session in identity provider has been opened at this point and that the client will therefore find it ready. The reply will redirect the client to the identity provider. The relying party will then wait for a notification about the result of the authentication attempt, hence the input choice on the operations `authSucceeded` and `authFailed`, which correlate through `secureToken`.

We now show the identity provider behavioural code sketch omitting the interface definitions: we assume input types to have their operation name with an initial uppercase letter.

```
cset { relyingPartyIdentifier:
          OpenAuthentication.relyingPartyIdentifier
          Authenticate.relyingPartyIdentifier,
       token: OpenAuthentication.token Authenticate.token }
main {
  openAuth( openRequest ); authenticate( authRequest );
  /* ... verify authentication ... */
  message.secureToken = openRequest.secureToken;
  if ( verified ) { authSucceeded@RelyingParty( message ) }
  else            { authFailed@RelyingParty( message ) }
}
```

The service can start a session with an input on `openAuth` (to be called by the relying party). The operation receives the values for initialising the correlation set, which is composed by two variables: `relyingPartyIdentifier` and `token`. We need both variables because there may be multiple active sessions for handling requests from different relying parties: two relying parties may generate a same value for `token`. We solve this issue by adding the identifier, e.g. a URL, of the relying party to the correlation set. After the session has been opened, we wait for the user credentials on operation `authenticate`. The credentials are verified and the result sent to the relying party.

[2] Aliasings for `clientToken` are left unspecified in the relying party implementation sketch, since it will only be used after establishing whether the user can log in.

7 Related Work and Conclusions

Related Work. Previous versions of JOLIE (including SOCK [7]) feature correlation sets where correlation data is manipulated within sessions. However, they support no correlation aliasing and no static analysis for identifying bad correlation programming. Even though SOCK features the Request-Response pattern, its semantics does not meet our requirement of integration since the reply is not routed through a private channel. Instead, it is correlated again to the session of the invoker, thus making it similar to an interaction performed by means of two One-Way operations. Moreover, they do not feature multiple correlation sets. All correlation variables are, instead, put in one single correlation set, which does not act as unique session identifier: sessions may be ambiguous under correlation and the related message routing non-deterministic.

Our approach takes inspiration from BPEL [17], which supports multiple correlation sets for identifying sessions. In BPEL, correlation programming is mixed with that of behaviours. Correlation sets are scoped in specific code blocks, and different input activities can use different correlation sets for receiving even if they use the same operation. This makes BPEL programming more error-prone than in our approach, where correlation sets are based on the service interface (its operations) and defined independently from the behaviour. Our language expressiveness is still high, due to correlation data manipulation inside sessions. BPEL does not support correlation programming with a typing discipline, but relies on run-time faults for signaling undesired situations that the programmer specifies manually. Finally, BPEL does not come with formal specifications, leaving much of the burden in handling the complexity of a distributed system to the programmer and the interpreter implementations.

Blite [13] is a model for service orchestration, whose programs can be compiled to BPEL processes [4]. The model is formal, but the final compilation to BPEL makes the approach suffer from the unpredictable behaviour of the execution engine, due to the lack of formality of BPEL specifications. Similarly, the calculus for web services COWS [12] allows to correlate sessions based on channel usage. COWS features several tools for static analyses and an interpreter, however it lacks a fully-fledged language implementation.

[10] provides an implementation of channel-based sessions relying on session types [9, 8]. In this setting, message routing does not rely on data transmission.

Conclusions. We have presented a language for programming services with correlation sets, investigating the interplay between some key aspects of SOC and correlation-based programming. Our approach features a direct manipulation of correlation data in programs and a notion of correlation aliasing. We have shown how both aspects can be disciplined by means of a type system. The applicability of our work has been demonstrated by exposing implementations of real-world scenarios where correlation sets can be successfully employed. Our solution has replaced the previous correlation mechanism in the JOLIE language. The features guaranteed by properties 2 and 3 are similar to those provided by private channels in the π-calculus. In our approach different sessions use different instances of correlation sets, much like in the π-calculus replications of a same process use different private channels.

Our semantics for message queues can lead to deadlocks, because a session must consume messages in the same order in which they are received. There

are various potential solutions to this problem. For instance, each session could manage a separate queue for each correlation set, or for each operation. Another issue in our model is that it does not handle session garbage collection, i.e. terminated sessions are not removed from their executing service. Handling this aspect is nontrivial, because a terminated session may have some messages left in its queue which must be dealt with. We leave the investigation of these issues to future work, as they are not relevant for the results presented in this paper.

More complex forms of analysis may be developed for correlation. An interesting aspect would be to analyze the behaviour of service networks by introducing behavioural types for participants such as session types [8]. Another topic to be explored is that of security. Programs may be checked to establish that correlation values are not *compromised*.

References

1. Internet Relay Chat Protocol. http://tools.ietf.org/html/rfc1459.
2. On-line appendix. http://www.itu.dk/people/fabr/icsoc2011.
3. OpenID implementation. http://www.jolie-lang.org/files/icsoc2011/openid.zip.
4. L. Cesari, A. Lapadula, R. Pugliese, and F. Tiezzi. A Tool for Rapid Development of WS-BPEL applications. In *SAC*, pages 2438–2442, 2010.
5. Free Software Foundation (FSF). GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html.
6. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic Error Handling in Service Oriented Applications. *Fundamenta Informaticae*, 95(1):73–102, 2009.
7. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proc. of ICSOC 2006*, pages 327–338, 2006.
8. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL'08*, volume 43(1), pages 273–284. ACM Press, 2008.
9. Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.
10. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541, 2008.
11. JOLIE. JOLIE: Java Orchestration Language Interpreter Engine. http://www.jolie-lang.org/.
12. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 33–47, 2007.
13. A. Lapadula, R. Pugliese, and F. Tiezzi. A Formal Account of WS-BPEL. In *Proceedings of COORDINATION 2008*, pages 199–215, 2008.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
15. F. Montesi, C. Guidi, I. Lanese, and G. Zavattaro. Dynamic Fault Handling Mechanisms for Service-Oriented Applications. In *Proceedings of ECOWS 2008*, pages 225–234, 2008.
16. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proceedings of ECOWS 2007*, pages 13–22, 2007.
17. OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/.
18. OpenID. OpenID Specifications. http://openid.net/developers/specs/.
19. World Wide Web Consortium (W3C). SOAP Specifications. http://www.w3.org/TR/soap/.
20. World Wide Web Consortium (W3C). Web Services Description Language. http://www.w3.org/TR/wsdl.