# Implementing Session Centered Calculi[⋆]

Lorenzo Bettini[1], Rocco De Nicola[2], Michele Loreti[2]

[1]Dipartimento di Informatica, Università di Torino.
[2]Dipartimento di Sistemi e Informatica, Università di Firenze.
{bettini,denicola,loreti}@dsi.unifi.it

**Abstract.** Recently, specific attention has been devoted to the development of service oriented process calculi. Besides the foundational aspects, it is also interesting to have prototype implementations for them in order to assess usability and to minimize the gap between theory and practice. Typically, these implementations are done in Java taking advantage of its mechanisms supporting network applications. However, most of the recurrent features of service oriented applications are re-implemented from scratch. In this paper we show how to implement a service oriented calculus, CaSPiS (Calculus of Services with Pipelines and Sessions) using the Java framework IMC, where recurrent mechanisms for network applications are already provided. By using the session oriented and pattern matching communication mechanisms provided by IMC, it is relatively simple to implement in Java all CaSPiS abstractions and thus to easily write the implementation in Java of a CaSPiS process.

## 1 Introduction

Service-oriented computing is calling for novel computational models and languages and recently specific attention has been devoted to the development of service oriented process calculi that can lay the basis for analyzing and experimenting with components interactions, safe service composition, and for formalizing and reasoning about aspects of service level agreements. Recently, many calculi have been proposed and most of them are based on process algebras enhanced with mechanisms for describing safe client-service interactions and with operators for composing services. Besides the foundational aspects, it is also interesting to have prototype implementations of these calculi, in order to assess their practical usability and to minimize the gap between theory and practice.

In this paper we show how to implement a service oriented calculus, CaSPiS (Calculus of Services with Pipelines and Sessions) [3] using a generic Java framework called IMC (*Implementing Mobile Calculi*) where recurrent mechanisms for network applications are already provided. CaSPiS is the evolution of SCC (Serviced Centered Calculus) [2], a calculus for services, that stemmed from a coordinated effort within the EU funded project SENSORIA [12] that aims at developing a novel, comprehensive approach to the engineering of software systems for service-oriented computing.

⋆ The work presented in this report has been partially supported by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09) by the MIUR project EOS DUE.

IMC was prompted by the growing number of experiments on process calculi and by the need of easing the implementation phase and was used as a kind of middleware for different distributed calculi [1]; it provides the necessary tools for implementing the run-time system of new languages directly based on distributed calculi (possibly with code mobility). The aim of IMC was to enable the implementer of a new language to concentrate on the parts that are really specific of the considered system, and to rely on the framework for standard mechanisms for distribution and mobility.

IMC provides means for transparent code mobility, for building communication protocols by composing sub-components dynamically and for managing node topology. All these mechanisms are rendered as abstract as possible to ease, e.g., switching from a specific communication protocol to another, without modifying the other parts of an application. IMC can be straightforwardly used if no specific advanced feature is needed; but a user can customize parts of the framework by providing its own implementations for the interfaces used in the package. Customizations can take advantage of design patterns such as *factory method*, *abstract factory*, *template method* and *strategy* [7] that are used throughout the packages.

CaSPiS [3] is a formalism useful for experimenting with service oriented calculi implementations, with advanced features and a clear theoretical foundation. It is *dataflow* oriented and makes use of a pipelining operator to model the exchange of information between sessions (sequences of structured communications between two peers). Services are seen as passive objects that can be invoked by clients and service definitions can be seen as specific instances of input prefixed processes. The two endpoints of a session can communicate by exchanging messages. A fresh shared name is used to guarantee that messages are exchanged only between partners of the same session, so that two instances of the same persistent service (that was invoked from two different sessions) run separately and cannot interfere. The central role assigned to sessions and the direct use of operators for modeling session interaction renders the logical structure of programs clearer and leads to a well disciplined service specification language that guarantees proper handling of session closures and in general simplifies reasoning on the specified services. The idea of session is not new. Indeed, in [10, 8] identifies a simple type-regulated interactions in $\pi$-like languages. The calculus makes use also of a new policy for handling (unexpected or programmed) session closures that in the original SCC calculus was somehow "rudimental". Indeed, in SCC closed session as well as nested subsessions are simply *terminated* and no information is sent to the counterpart. In CaSPiS new primitives are introduced for handling session closure and for reacting to an unexpected session closures.

The rest of the paper is organized as follows. Section 2 provides a brief overview of CaSPiS, Section 3 presents IMC while in Section 4 the actual implementation of CaSPiS is presented. Section 5 shows how the proposed implementation can be used for developing simple services. The final section contains an example of a CaSPiS program and some concluding remarks.

$$
\begin{array}{rlll}
P,Q ::= & \sum_{i\in I}\pi_i P_i & \text{Guarded Sum} & \mid \dagger(k) & \text{Signal}\\
\mid & s_k.P & \text{Service Definition} & \mid r \rhd_k P & \text{Session}\\
\mid & \bar{s}_k.P & \text{Service Invocation} & \mid \blacktriangleright P & \text{Terminated Session}\\
\mid & P > Q & \text{Pipeline} & \mid P|Q & \text{Parallel Composition}\\
\mid & \mathsf{close} & \text{Close} & \mid (\nu n)P & \text{Restriction}\\
\mid & k.P & \text{Listener} & \mid\, !P & \text{Replication}
\end{array}
$$

$$
\begin{array}{rll}
\pi ::= & (F) & \text{Abstraction}\\
\mid & \langle V\rangle & \text{Concretion}\\
\mid & \langle V\rangle^{\uparrow} & \text{Return}
\end{array}
\qquad
\begin{array}{l}
F ::= u \mid ?x \mid f(\widetilde{F})\\
V ::= u \mid f(\widetilde{V})
\end{array}
$$

**Fig. 1.** Syntax of full CaSPiS.

## 2 CaSPiS

CaSPiS (*Calculus of Services with Pipelines and Sessions*) [3] is a core calculus equipped with linguistic constructs for handling sessions and that relies on three main concepts:

1. service definition/invocation
2. bi-directional sessioning as a means for structuring client-service interaction
3. pipelining as a means of composing services.

The syntax of CaSPiS is in Figure 1. In the following we will comment the main constructs of CaSPiS, skipping the standard process algebras operators (such as, e.g., non deterministic choice $\sum_{i\in I}\pi_i P_i$, restriction $(\nu n)P$, parallel composition $P|Q$ and replication $!P$). Interested readers are referred to [3] for further details.

Within CaSPiS, service definitions and service invocations are rendered respectively as $s_{k_1}.P$ and $\bar{s}_{k_2}.Q$ where $s$ is a service name, $k$ is the *handler* used for managing session closures while $P$ and $Q$ implement the service and the client *protocols* respectively.

A service definition $s_{k_1}.P$ and a service invocation $\bar{s}_{k_2}.Q$ running in parallel can synchronize with each other. As a result, a new, private, session $r$ will be created. The session has two ends, one at the client's side where protocol $Q$ is running and one at the service's side where protocol $P$ is running. A value produced by a *concretion* at one side can be consumed by an *abstraction* at the other side.

A concretion $\langle V\rangle P$ can evolve to $P$ emitting value $V$. Dually, an abstraction $(F)P$ is a form of guarded command that relies on pattern-matching: $(F)P$ can evolve to $P\sigma$ retrieving value $V$, provided pattern $F$ matches value $V$. Here, the pattern-matching function *match* is defined as expected: $match(F,V)=\sigma$, if $\sigma$ is the (uniquely determined) substitution that permits identifying pattern $F$ and values $V$.

The return primitive $\langle V\rangle^{\uparrow}P$ can be used to return a value *outside* of the current session, if the enclosing environment is capable of consuming it. Sessions, service definitions and service invocations can of course be nested at arbitrary depth. No activity can take place under the scope of a dynamic operator (service definition or invocation, guarded sum, right-hand side term of a pipeline and replication). On the contrary, when considering non dynamic contexts, including sessions, concurrent activities can take place at any level of session nesting. Sessions do not constrain in any way actions that are not value production, consumption or return, that is, service invocation and silent steps.

CaSPiS is equipped with primitives for handling session closure. These primitives are useful to garbage-collect terminated sessions and, most importantly, to explicitly program session termination in order to manage abnormal events or timeouts.

Upon creation of a session, one associates with the session a pair of names, $(k_1,k_2)$, identifying a pair of *termination handlers* services, one for each side. Then:

1. a session side is terminated when its protocol executes the command $\mathsf{close}$;
2. right after execution of $\mathsf{close}$ a signal $\dagger(k)$ is sent to the *listener* on $k$ (such a listener will have the syntactic form $k.R$) running at the *opposite* side of the session.
3. at the same time, the session side that has executed $\mathsf{close}$ will enter a special closing state denoted by $\blacktriangleright P$, where all subsessions of $P$ will be gradually and automatically closed.

Information about termination handlers to be used is exchanged by the two sides at invocation time. Operational rules governing service synchronizations are the following:

$$s_{k_1}.P \xrightarrow{s(r)_{k_1}^{k_2}} r \rhd_{k_2} P \qquad\qquad \bar{s}_{k_2}.P \xrightarrow{\bar{s}(r)_{k_2}^{k_1}} r \rhd_{k_1} P$$

$$\frac{P \xrightarrow{s(r)_{k_1}^{k_2}} P' \quad Q \xrightarrow{\bar{s}(r)_{k_2}^{k_1}} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

Hence, process $\bar{s}_{k_1}.Q|s_{k_2}.P$ evolves to $(\nu r)(r \rhd_{k_2} Q|r \rhd_{k_1} P)$. There, if $Q$ terminates with $\mathsf{close}$, the termination handler $k_2$ of the callee will be activated. Vice versa, if $P$ terminates with $\mathsf{close}$ the termination handler $k_1$ of the caller will be activated:

$$\frac{P \xrightarrow{\mathsf{close}} P'}{r \rhd_k P \xrightarrow{\tau} \blacktriangleright P'|\dagger(k)} \qquad\qquad \blacktriangleright r \rhd_k P \xrightarrow{\tau} \blacktriangleright P|\dagger(k)$$

A typical behavior for a listener is that of closing the current session as soon as a signal $\dagger(k)$ is received. This listener can be rendered as: $k.\mathsf{close}$.

Processes can be composed by using the *pipeline* operator $P > Q$. Whenever $P$ produces a value $V$ that $Q$ can consume, a reduction will trigger a new instantiation $Q'$ of $Q$. After this reduction, $Q$ is again ready to consume the next value produced by $P$, if any:

$$\frac{P \xrightarrow{\langle V\rangle} P' \quad Q \xrightarrow{(V)} Q'}{P > Q \xrightarrow{\tau} (P' > Q) \mid Q'}$$

### 2.1 A small example

In this section we will show how CaSPiS can be used for modeling a simple system used for computing the basic arithmetic operations. In the example, and in the rest of this paper, we will sometime use standard programming language operators like selection (**if** – **then** – **else**) or iteration (**while** – **do**) that can be implemented as macros in CaSPiS. Service `calculator` can be implemented in CaSPiS as follows:

$$
\begin{array}{ll}
!(\nu k)\mathtt{calculator}_k. & !(\mathtt{"sum"}, ?x, ?y)\langle \mathtt{"result"}, x+y\rangle \\
& \mid\ !(\mathtt{"sub"}, ?x, ?y)\langle \mathtt{"result"}, x-y\rangle \\
& \mid\ !(\mathtt{"mul"}, ?x, ?y)\langle \mathtt{"result"}, x*y\rangle \\
& \mid\ !(\mathtt{"div"}, ?x, ?y)\ \textbf{if}\ y=0\ \textbf{then}\ \langle \mathtt{"fail"}\rangle\ \textbf{else}\ \langle \mathtt{"result"}, x/y\rangle \\
& \mid (\mathtt{"off"})\mathsf{close} \\
& \mid k.\mathsf{close}
\end{array}
$$

after service `calculator` is invoked, processes for managing basic arithmetic operations and those for controlling session termination are installed in the established session. The processes for arithmetic operations wait for tuples containing the operation to be computed (`"sum"`, `"sub"`, `"mul"` and `"div"`) and the two operands ($x$ and $y$) and then send to the callee the result. In case of a division operation, message `"fail"` is sent to the callee when $y$ is 0. Moreover, when message `"off"` is received the established session is closed. Finally, listener $k.\mathsf{close}$ is used for managing unexpected session closing by the client. To avoid interferences, name $k$ is private. Since this service is replicated, it is always available for the invocation.

Service `calculator` can be used, for instance, for computing *Greatest Common Divisor* between two integers using Euclid's algorithm. Service `gcd` is defined as follows:

$$
\begin{array}{l}
!\mathtt{gcd}.(?x, ?y) \\
\quad \textbf{if}\ (y=0)\ \textbf{then}\ \langle x\rangle \\
\quad \textbf{else} \\
\qquad \textbf{if}\ (x<y)\ \textbf{then}\ \overline{\mathtt{gcd}}.\langle y,x\rangle(?z)\langle z\rangle^{\uparrow} \\
\qquad \textbf{else}\ P > (?u, ?w)\overline{\mathtt{gcd}}.\langle u,w\rangle(?z)\langle z\rangle^{\uparrow}
\end{array}
$$

where process $P$ is defined as follows:

$$P \triangleq (\nu k')\overline{\mathtt{calculator}}_{k'}.\langle \mathtt{"sub"}, x, y\rangle(\mathtt{"result"}, ?z)\langle z, y\rangle^{\uparrow}\langle \mathtt{"off"}\rangle | k'.\mathsf{close}$$

## 3 The IMC framework

We now sketch the main functionalities and classes of the framework, for further details we refer to the IMC web page `http://imc-fi.sf.net`). IMC consists of three main subpackages: `protocols`, `mobility` and `topology` that deal with communication protocols, code mobility and network topology, respectively. Since mobility is not employed in CaSPiS, we will ignore this subpackage in the following description.

IMC provides tools to define customized protocol stacks, which are viewed as a flexible composition of micro-protocols, and permits achieving adaptable forms of communication transparency, which are needed when implementing an infrastructure for global computing. In IMC, a *network protocol* is viewed as an aggregation of *protocol states*: a high-level communication protocol can indeed be described as a state automaton. Thus, the programmer must simply provide the implementation of each state, put them in a protocol instance, and then start the protocol. The protocol states abstract away from the specific communication layers. This permits re-using protocol implementations independently from the underlying communication means: the same protocol can then be executed on a TCP socket, on UDP packets or even on streams attached to a file (e.g., to simulate a protocol execution). This abstraction is implemented by specialized streams: `Marshaler` (for writing) and `UnMarshaler` (for reading). These streams provide high-level and encoding-independent representations of information to be sent or received.

The data in these streams can be "pre-processed" by some customized *protocol layers* that remove some information from the input and add some information to the output: typically this information is a header removed from the input and added to the output. The base class `ProtocolLayer` deals with these functionalities, and can be specialized by the programmer to provide his own protocol layer. These layers are then composed into a `ProtocolStack` object that ensures the order of preprocessing passing through all the layers in the stack. Each layer is independent and the composition of layers in a protocol stack takes place at run-time. For instance, the programmer can add a layer that removes a sequence number from an incoming packet and adds the incremented sequence number into an outgoing packet.

In IMC a participant in a network is an instance of the class `Node` of the package `topology`. A node is also a container of running processes that can be thought of as the computational units. The framework provides all the means for a process to access the resources contained in a node and to migrate to other nodes. A process is an instance of a subclass of the class `NodeProcess`, and can be added to a node for execution with the method `addProcess` of the class `Node`. A node keeps track of all the processes that are currently in execution and handles their termination when the node itself is terminated. The entry point of a `NodeProcess` is the abstract method `execute` that must be implemented in subclasses of `NodeProcess`. Actually, a process can interact with the node it is running on only through a `NodeProxy`, which ensures security by restricting the node interface visibility to a subset.

The framework provides classes and protocols to deal with *sessions*, a base concept of service calculi. The concept of session is logical, since it can then rely on a physical connection (e.g., TCP sockets) or on a connectionless communication layer (e.g., UDP packets). A `SessionManager` instance will keep track of all the sessions. This can be used to implement several network topology structures. A `Session` instance is identified by two `SessionId` objects, one indicating the local end and the other one indicating the remote end. A `SessionId` contains information about the "location" or "address" of a node; this concept depends on the specific communication medium: for instance, for an IP communication it will be a string of the shape IP:port. Moreover, it contains information about the specific low level communication protocol. For instance, `"udp-myhost.com:9999"` represents a UDP communication with the host `"myhost.com"` on port 9999. Upon establishing a session, the `SessionId` is used to determine the low level communication layer. Thus, switching from a communication layer to another is only a matter of changing the `SessionId`, while all the other classes in IMC are independent from this, and do not need to be changed. A `Session` can be established by using the method `connect`, of class `Node`, specifying the `SessionId` of the remote end; a session request can be accepted by using the method `accept`, by specifying the local `SessionId`. These methods return a `ProtocolStack` object (where the lowest layer is already set as explained above); this can then be customized by adding specific `ProtocolLayer` objects. Finally it can be passed to a `Protocol` instance that will run upon it.

IMC provides an implementation of *tuples*, *tuple spaces* and the associated *pattern matching* retrieval mechanism, thus, the programmer can use the *generative* and asynchronous communication mechanisms typical of *Linda* [9]. Notice that the implementation of tuple spaces in IMC also provides extended operations such as non-blocking retrieval operations, and retrieval operations that permit reading/removing any tuple (without specifying its template). Furthermore, there is also a blocking version of the **out** operation: this permits implementing a synchronous communication mechanism still relying on pattern matching (this will be the case of the communication in CaSPiS, Section 4).

Inside IMC inter-objects communication takes place via the *event* based functionalities provided by IMC. In particular, most classes of the framework are endowed with event generation capabilities (e.g., `ProtocolState`, `ProtocolLayer`, `Node`, etc.). This permits keeping the classes loosely coupled and communications among objects in the framework highly flexible. It is then easy to intercept, e.g., new connection requests, connection failures and session closures. With this respect, the framework notifies the processes involved in a session about the closure of the session so that they can perform finalization operations.

## 4 JCaSPiS: CaSPiS implementation in IMC

In this section we present JCaSPiS: a Java framework that permits implementing service oriented applications based on CaSPiS paradigm. Notice that, CaSPiS operators like parallel composition and restrictions, can be directly implemented in Java. Indeed, the former is implemented by using *threads*, while the latter is obtained by considering the creation of *new objects* like, for instance, the instantiation of *new services*. Other operators, like *non deterministic choice* and *replication* are implemented in JCaSPiS in a restricted way. Indeed, in JCaSPiS we will consider only the *choice* between *input* actions while *replication* will be available only on service definitions.

The implementation of other JCaSPiS primitives requires more attention. Indeed, to allow JCaSPiS programs to interact with existing services, implementation of *service definitions*, *service invocations* and *sessions* has to take into account existing protocols and technologies for services. IMC provides the Java classes that can be easily used for handling connections and disconnections among nodes over a network. JCaSPiS specializes these classes in order to handle Service Oriented Protocols. Two kinds of connection protocols are considered: TCP and HTTP. The former is already provided by IMC framework, while the latter has been implemented by using Simple Web Server

[13], a Java library, released under the GNU LGPL, providing an extensible HTTP engine.

Three protocols for services interactions have been developed: BYTE_CODE, XML and SOAP. The first one, is used when service interaction is implemented by serializing Java objects. XML and SOAP protocols are used when service interaction is based on XML and SOAP messages respectively. Thanks to the modularity of IMC, new service interaction protocols, as well as new implementations of the one already available, can be easily integrated within JCASPIS.

Provided protocols are implemented as new layers that permit marshaling/unmarshaling data as Java objects or within XML and SOAP messages respectively. BYTE_CODE protocol is directly developed over the existing Java serialization mechanisms while XML and SOAP implementations are based on two standard J2EE [5] libraries, *Java API for XML Binding* (JAXB) [6] and *SOAP with Attachments API for Java* (SAAJ) [14].

*Java API for XML Binding* is a Java library that permits mapping Java classes to XML representations. Indeed, by using JAXB Java objects can be marshaled into XML and vice-versa. In other words, JAXB permits sending and receiving Java objects in XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure. JAXB is one of the APIs in the Java EE platform, and is part of the Java Web Services Development Pack (JWSDP). It is also one of the foundations for WSIT. JAXB is part of SE version 1.6.

*SOAP with Attachments API for Java* (SAAJ) provides primitives for producing and consuming messages conforming to the SOAP specification and with attachments. Indeed, SAAJ automates many of the required steps for creating/analyzing SOAP messages.

In the following we will describe how key notions of CaSPiS are implemented within IMC.

*Services* Services are referenced by means of a `Service` object that contains *service name*, the `SessionId`, which is used for identifying the connection protocol and the address of the service, and the protocol used for service interaction. For instance, service `pair` running at host `test.unifi.it:8080` based on XML messages is referenced as:

s = **new** *Service*(**new** *IpSessionId*(`"test.unifi.it"`, 8080), `"pair"`, `"xml"`);

When a service is invoked, a connection to the remote host providing the requested service is established. Moreover, the protocol `ProtocolStack` implementing the required conversation protocol (i.e., IMC, XML or SOAP) is instantiated and stored within an object instance of class `Connection`. This object, which abstracts from a specific interaction protocol, is used for implementing service interactions.

*Processes* CaSPiS processes are implemented by classes inherited by the abstract class `Process`. The classes derived from `Process` must provide the implementation of the entry point method `execute`, and can use all the methods for exchanging data through a session and outside the session environment, and for publishing or invoking a service. For instance, CaSPiS process $(?x)(?y)\langle x,y\rangle$, that emits a pair containing two read values, will be rendered in JCASPIS as follows:

```
public class PairServiceProcess extends SessionProcess {
  public void execute() throws IMCException {
    Object first = inAction(); // accept any template
    Object second = inAction(); // accept any template
    send(new Tuple(first, second));
  }
}
```

Similarly, the process $(\langle\texttt{"a"}\rangle|\langle\texttt{"b"}\rangle|(?x)\langle x\rangle^{\uparrow})$, that sends two values, retrieves a pair and emits it in the enclosing context, can be implemented as follows:

```
public static class PairClient extends ParallelProcess {
  public void execute() throws IMCException {
    outAction(new Tuple("a"));
    outAction(new Tuple("b"));
    Tuple t = (Tuple) inAction();
    returnAction(t);
  }
}
```

Each process can execute method `runProcess(Process p)` for activating the execution of process `p`.

*Contexts* `Process` instances are executed within a `Context`. Abstract class `Context` provides the following methods for:

- publishing services that will be invoked by remote partners (`publish`);
- invoking remote services and instantiating local processes implementing service interactions (`call`);
- executing basic CaSPiS actions (`inAction`, `outAction` and `returnAction`), for verifying whether an action can be executed (methods `checkIn`, `checkOut` and `checkReturn`) and for closing the enclosing session (`close`).

`Contexts` can be nested. For this reason `Context` also keeps track of all its nested components. By using the IMC mechanisms to react upon session closing, it automatically forwards the session closing operation to all its nested `Contexts`.

*Service Publication and Invocation* A service is published by invoking one of the following methods on a context:

- `publish(Service s, Process p)`
- `publish(Service s, Class<? extends Process> c)`
- `publish(Service s, Class<? extends Process> c, boolean per)`

These methods publish a service within the current node. When a request for the published service is received, process `p` (or an instance of class `c`) is activated. Boolean parameter `per` is used for determining if the service is persistent, namely if the service is still available after the first invocation.

The following code permits publishing the service `s` defined above:

*publish*(s, PairServiceProcess.**class**, **true**)

A service can be invoked by executing one the following methods:

- `call(Service s, Process p)`
- `call(Service s, Class<? extends Process> c)`

when one of these methods is invoked, a connection to the remote host is opened and a `Session` (described in the following) is installed within the actual context. Process `p` (or an instance of class `c`) is executed within the new session.

To invoke service `s`, the following code is executed:

*call*(s, **new** *PairClient*(), **true**)

The implementation of publish and call methods in `Context` rely on abstract methods:

- `Connection publish(ServiceName s)`
- `Connection call(ServiceName s)`

The former waits for a request for service `s`, the latter establishes a connection with the remote host providing service `s`. Both methods return an instance of `Connection` used for interacting with the caller/callee. Hence, a new session is created and the obtained object is used for interacting with the remote participant.

JCaSPiS provides three implementations of abstract class `Context`: `Execution-Environment`, `Session` and `PipeLine`. They define a top level context, a session and a pipeline, respectively.

Class `ExecutionEnvironment` is also devoted to wait for incoming connections. Indeed, its constructor is parametrized with respect to the Internet addresses used for handling incoming TCP and HTTP connections. Moreover, `ExecutionEnvironment` keeps track of the published services. These are stored within a `ServiceRegistry`. When a connection request is received, this object is used for determining the process that has to handle the received service request. Since the same service can be published with different implementations, `ServiceRegistry` can be specialized for implementing different service selection policies. At the moment, implementations of a service are collected in a list, and the first available is selected.

Classes `Session` and `PipeLine` provide an implementation for CaSPiS sessions and pipelines respectively. `Sessions` are installed within a context when a service is invoked. Interactions with the remote participants are performed via an instance of `Connection` that contains a reference to the `ProtocolStack` that is created once the connection is established.

CaSPiS pipelines are implemented by means of `PipeLine`. This class contains a list of `Abstractions`. These are processes parametrized with respect to a `Template`, i.e., an object that permits selecting received messages. Output actions executed by running processes are intercepted in order to activate the processes corresponding to a matching template.

In Figure 2 we report the class diagrams of JCaSPiS classes described in this section. Notice that `Process`, `PipeLine` and `Session` implement interface `Activity`. This is the interface that characterize objects that can be executed within a `Context`.

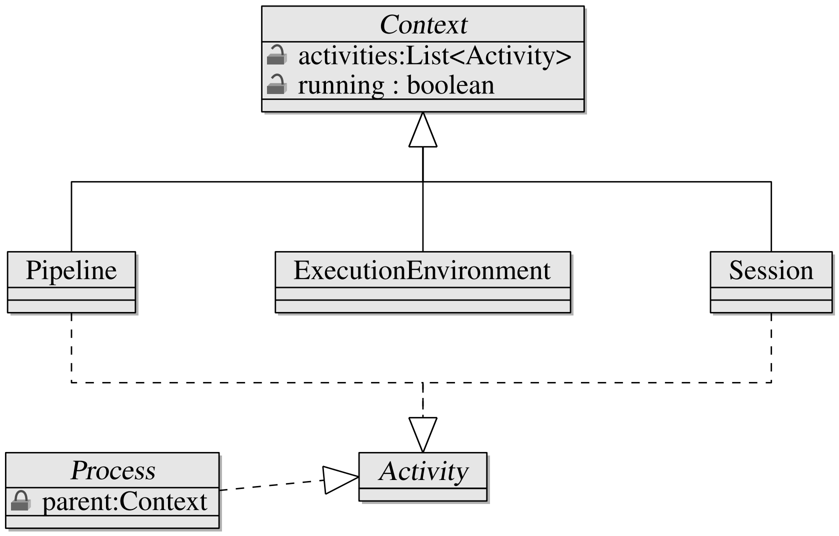

**Fig. 2.** JCASPIS: Class Diagram

*Session Interactions* The communication mechanism in CaSPiS is based on structured values and pattern matching, thus, we will use the tuple space based communication provided by IMC. The retrieve operation can be performed using the method `inAction`, that can also accept as a parameter a template, which extends abstract class `Template`, specifying the data we are willing to receive; the write operation can be performed using the method `outAction`, that takes as a parameter the message we want to send. Finally, the method `returnAction` is used for sending a value just outside the current session. This method is implemented by invoking method `outAction` of the enclosing context.

Classes `Session` and `PipeLine` provide different implementations for `inAction` and for `outAction` methods. `Session` sends and receives messages over the corresponding service connection. `PipeLine` delegates input to the enclosing context while catches output for activating a process that will handle the sent message.

A session is closed when method `close` is invoked. Afterwards, remote connection and all the nested sessions are closed. Notice that, each action performed within a session that is terminating, or within one of its subsessions, leads to an exception. Moreover, to handle unexpected connection closures, each session is equipped with a *termination handler*. This is a process that is executed for handling proper session closure. Termination handler is associated to a session when service is invoked/published. Indeed, there are methods `call`/`publish` described above that take as an extra parameter the process or the process class to use as termination handler.

Methods for session interactions cannot be invoked on a top level context or when a session is closing. For this reason, `Context` also provides methods that can be used for verifying whether an action can be executed: methods `checkIn`, `checkOut` and `checkReturn`.

## 5 Implementing simple services with JCASPIS

In this section we will show how JCASPIS can be used for implementing simple services. In particular, we consider the implementation of the example presented in Section 2.1.

*Service calculator.* The first step for implementing a service in JCASPIS is to define the messages used for interacting with it. In the case of `calculator`, we will consider three classes for implementing the interaction messages:

– `Operation`, containing a reference to an operation (`op`) and the arguments (`x` and `y`);
– `Result`, containing the operation result (`result`);
– `Failure`, containing a text indicating the occurred error;

in correspondence of these, we have also to implement the templates used for retrieving expected messages[1]. The actual calculator service relies on:

– `OperationTemplate`, for matching `Operation` messages;
– `ResultTemplate`, for retrieving results;
– `FailureTemplate`, for intercepting computational failure.

The body of the service `calculator` can be rendered in JCASPIS as follows:

```
runProcess(new DivProcess());
runProcess(new SubProcess());
runProcess(new SumProcess());
runProcess(new MulProcess());
inAction(new StringTemplate("off"));
close();
```

This process first activates sub-processes for computing the arithmetic operations (Java code for `SumProcess` and `DivProcess` is reported in Listing 1) and then waits for a termination string (`"off"`) for closing the actual session. This message is retrieved using a `StringTemplate`. This is a template that matches only strings that are equal to the one passed to the constructor.

Notice that in this case we do not install any handler for intercepting unexpected session closure. Indeed, default termination handler is used. This is a process that automatically closes current session as soon as the remote participant terminates.

Since service calculator is based on a persistent session (many messages can be exchanged over the established connection), service calculator is published by using TCP as connection protocol and BYTE_CODE as interaction protocol:

[1] Classes for implementing values exchanged after a service invocation and templates to be used in the established session could be generated automatically from a standard XML textual representation like, for instance, WSDL.

```
public class SumProcess extends Process {
  public void execute() throws InterruptedException {
    while (true) {
      Operation op = inAction(new OperationTemplate(Operation.Type.SUM));
      outAction(new Result(op.getX()+op.getY()));
    }
  }
}

public class DivProcess extends Process {
  public void execute() throws InterruptedException {
    while (true) {
      Operation op = inAction(new OperationTemplate(Operation.Type.DIV));
      if (op.getY()==0) {
        outAction(new Failure("Division by 0!"));
      } else {
        outAction(new Result(op.getX()/op.getY()));
      }
    }
  }
}
```

**Listing 1:** Processes for handling `sum` and `div` operations

```
ExecutionEnvironment env = new ExecutionEnvironment(8080, 9000);
Service calc = new Service();
calc.setName("calculator");
calc.setSessionId(new IpSessionId("localhost", 8080));
calc.setLanguage("BYTE_CODE");
env.publish(calc, CalculatorProcess.class, true);
```

*Greatest Common Divisor.* Service `calculator` can be used for computing the *Greatest Common Divisor* between two integers. This service, named `gcd`, operates on two kinds of messages:

– `Pair`, which contains the values for which we want to compute GCD;
– `Result`, which contains the computed GCD.

Protocol of service `gcd` is implemented in JCaSPiS as follows:

```
Pair p = inAction(new PairTemplate());
int x = p.getX();
int y = p.getY();
if (y==0) {
  outAction(new Result(x));
}
if (x < y) {
  call(gcd, new RequestResponseProcess(new Pair(y,x)));
} else {
  Match m = new Match();
  m.add(new IntegerTemplate(), GcdAbstraction.class);
  pipeline(new CalculatorClient(new Operation(Operation.SUB, x, y)), m);
}
```

where `RequestResponseProcess`, which implements the standard request-response service interaction pattern, is defined as follows:

```
public class RequestResponseProcess extends Process {
  Object request;
  public RequestResponseProcess(Object message) {
    this.request = message;
  }
  public void execute() throws InterruptedException {
    outAction(request);
    Object response = inAction();
    returnAction(response);
  }
}
```

while abstraction `GcdAbstraction` is defined as follows:

```
public class GcdAbstraction extends Abstraction {
  public void execute() throws InterruptedException {
    runProcess(new RequestResponseProcess(getActivationMessage()));
  }
}
```

To guarantee interactions with existing web services, service `gcd` is developed over HTTP and SOAP:

```
ExecutionEnvironment env = new ExecutionEnvironment(8080, 9000);
Service gcd = new Service();
gcd.setName("gcd");
gcd.setSessionId(new HttpSessionId("localhost", 8080));
gcd.setLanguage("SOAP");
gcd.setServicePackage("org.cmg.caspis.ex.gcdulator:org.cmg.caspis.ex.gcd");
env.publish(gcd, GcdProcess.class, true);
```

## 6 Conclusions

The implementation of a language based on a process calculus typically consists of a run-time system (a sort of abstract machine) implemented in a high level language like

Java, and of a compiler that, given a program written in the programming language based on the calculus, produces code that uses the run-time system above. In this paper we have illustrated, by means of a case study, a possible methodology to accelerate the development of prototype implementation of such a run-time system, by relying on the IMC framework. In particular, we have described JCASPIS: the implementation of CaSPiS, a calculus that has recently been proposed within the EU project SENSORIA. The use of IMC has permitted accelerating the development of prototype implementations while concentrating only on the features that are specific of CaSPiS. Indeed, JCASPIS composed only by 43 classes and about 1700 lines of code. These classes provide 289 methods, and the average number of lines per method is 2.5.

*Implementing other session based calculi* JCASPIS can be easily extended to implement two other session based calculi that, like CaSPiS, have directly stemmed from SCC [2], namely SSCC [11] and Conversation Calculus [4]. Notice that, implementation of SSCC and CC would completely reuse large part of the JCASPIS framework.

SSCC is *stream* oriented with primitives for inserting/retrieving data in/from streams. Streams have been easily implemented in IMC by using classes for handling tuple spaces. The interface of `Process` and `Context` can be extended in order to consider method `feed` that is used for inserting a value inside a stream. The new context `Stream` has been introduced for collecting the values produced by the inner activities.

The Conversation Calculus (CC) has explicit and distinct message passing primitives to model inter and intra session communication. These primitives are based on *communication directions* (see [4]). To implement these primitives, `Process` as well as `Context` have been extended to consider communication directions.

As a future work, we plan to develop a high level programming language that, inspired by CaSPiS, could be used for programming services and to orchestrate existing ones. Given a program written in the programming language based on the calculus it will be translated in a Java program that uses JCASPIS classes. One of the advantages of this approach is that programs could be verified by using formal tools that are being developed for CaSPiS.

## References

1. L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *Proc. of 5th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*, volume 3543 of *Lect. Notes in Comput. Sci.*, pages 181–193. Springer, 2005.
2. M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a service centered calculus. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer, 2006.
3. R. Bruni, M. Boreale, R. De Nicola, and M. Loreti. Sessions and pipelines for structured services programming. In *Proc. of FMOODS'08*, Lect. Notes in Comput. Sci. Springer, 2008. To appear.
4. L. Caires and H. Viera. A note on a model for service oriented computation. In *ESOP '08*. Springer, 2008. To appear.

5. J. . E. Edition. `http://java.sun.com/javaee/`.
6. J. A. for XML Binding. `https://jaxb.dev.java.net/`.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. S. J. Gay and M. J. Hole. Types and subtypes for client-server interactions. In *Proc. of ESOP'99*, volume 1576 of *Lect. Notes in Comput. Sci.*, pages 74–90. Springer, 1999.
9. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
10. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 122–138. Springer, 1998.
11. I. Lanese, F. Martins, A. Ravara, and V. Vasconcelos. Disciplining orchestration and conversation in service-oriented computing. In *SEFM '07*, pages 305–314. IEEE Computer Society Press, 2007.
12. Sensoria Project. Public web site. `http://sensoria.fast.de/`.
13. Simple Web Server. `http://simpleweb.sourceforge.net/`.
14. S. with Attachments API for Java. `https://saaj.dev.java.net/`.