Multiparty sessions in SOC*

Roberto Bruni[1], Ivan Lanese[2], Hernán Melgratti[3], and Emilio Tuosto[4]

[1] Dipartimento di Informatica, Università di Pisa, bruni@di.unipi.it
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, lanese@cs.unibo.it
[3] IMT, Alti Studi Lucca, Lucca, hernan.melgratti@imtlucca.it
[4] Department of Computer Science, University of Leicester, et52@mcs.le.ac.uk

Abstract. Service oriented applications feature interactions among many participants over the network. Mechanisms such as correlation sets and two-party sessions have been proposed in the literature to separate messages sent to different instances of the same service. This paper presents a process calculus featuring dynamically evolving multiparty sessions to model interactions that spread over several participants. The calculus also provides primitives for service definition/invocation and for structured communication in order to highlight the interactions among the different concepts. Several examples from the SOC area show the suitability of our approach.

1 Introduction

Service oriented computing (SOC for short) is emerging as the main paradigm to distribute computation over the Web, by composing services made available by different organizations. It relies on W3C standards such as SOAP [15] (transmission), WSDL [17] (service description), UDDI [12] (service discovery), etc. for disciplining the interactions among heterogeneous applications.

Since many instances of the same computation can be carried on at the same time (e.g., different clients trying to book flights from the same airline), it is important to guarantee that the different interactions do not interfere, and messages are routed to the intended recipient. In current technologies, interactions are quite basic (e.g., WSDL one-way and request-response modalities). Emerging standards like WS-BPEL [13] and WS-CDL [16] exploit the idea of *correlation sets* to model more complex interaction patterns. Correlation sets allow to route a message to a specific instance of the invoked service based on a pre-defined subset of the invocation parameters (e.g., requests referring to the same username are routed to the same session). On the one hand, this guarantees a good expressive power without requesting too much help from the underlying middleware. On the other hand, the emerging patterns of interaction are quite difficult to analyze, since they depend on data values, and each application on the Web can interfere with a session provided that it knows (or it uses by chance)

* Research supported by the Project FET-GC II IST-2005-16004 Sensoria and the UK project HiDeA4SOC.

the right values. This makes the formal specification and analysis of nowadays Web service-based applications very difficult. This is a big drawback, considering that development of Web applications is an error-prone activity (since it concerns complex interactions among distributed components), and one would like to have safe development techniques.

In the literature on formal methods for Web services the concept of a *session* has been proposed as a solution to this problem [1, 8]. In this setting a session encloses an arbitrarily complex interaction among two partners, and it guarantees that the messages are routed as desired during the whole conversation, without interferences from the outside. From this point of view, correlation sets can be seen as a particular implementation of the concept of a session.

Since modern service oriented applications are programmed on top of the TCP/IP stack, it is usually accepted that sessions involve only two participants (typically, one playing the role of a client and the other the role of a server). On the contrary, many applications would more naturally be modeled via protocols allowing *multiparty sessions*, namely sessions to which more than two actors can take part. For example, consider an online gaming application, where a server provides the playing platform and users can log into different games. While logged in, users in the same game room can interact by following a game-specific protocol, while users in other rooms can not interfere. This is modeled in a very natural way by considering a different multiparty session for each room.

In this paper we propose $\mu\mathsf{se}$ (read "muse", after "MUlti-party SEssion" calculus), a process calculus designed for modeling multiparty sessions in an abstract way. Our aim is to propose a model based on multiparty sessions, and to analyze the relations between sessions and the other main features of SOC: services, communication, sites, etc. To this end we consider a world partitioned into sites, where services, sessions and processes live. Services can be dynamically published. Sessions are dynamically created and new participants can join them at runtime (concurrent ongoing sessions can also be merged). Communication is based on two basic distinct mechanisms: intra-session communication and intra-site communication. Intra-session communication models communications between different endpoints of the same session, to be realized by exploiting the underlying middleware (TCP/IP, SOAP, ...). Intra-site communications concern instead processes in the same site, and model local communication, to be realized by exploiting e.g. the shared memory of the machine. We equip $\mu\mathsf{se}$ with a (weak) bisimilarity-based equivalence whose appropriateness is illustrated by means a small proxy scenario.

From a technical point of view, $\mu\mathsf{se}$ is based on the experience maturated inside the Sensoria project (on Software Engineering for Service-Oriented Overlay Computers) [14], and takes inspiration from π-calculus [11] for the basic communication mechanisms (i.e., it is based on message passing) and on the calculi developed inside Sensoria for the more service oriented features. The main novelty of our calculus w.r.t. the last ones are multiparty sessions. However this has a strong impact also on other features. For instance, while in binary sessions the intended recipient of an intra-session communication is always understood (the other endpoint of the session), in multiparty sessions many recipients are possible, and an additional coordination mechanism is needed.

Structure of the paper: The μse calculus is introduced in § 2: § 2.1 provides an informal description of the main elements of the calculus, while § 2.2 and § 2.3 formally define the language. The coding of several interaction patterns is given in § 3. § 4 proposes an observational semantics for μse based on the standard notion of weak bisimulation. Final remarks are drawn in § 5.

2 The μse calculus

This section introduces the μse calculus. The main features of μse are (i) nested multi-party and dynamically joinable sessions, (ii) intra-session and intra-site communications, (iii) dynamic service publication.

2.1 A μse walkthrough

μse has been designed so to keep a clear conceptual distinction, even at the syntax level, between different concerns distilled from the SOC paradigm. This allows for an incremental presentation of the calculus that can serve to emphasize also the interplay between the various features considered.

Services and multiparty sessions. The kernel syntax of μse includes ordinary operators such as the nil process $\mathbf{0}$, parallel composition $P|Q$ and name restriction $(\nu n)P$, together with primitives for service definition, for service invocation, for enclosing a process in a session and for dynamically install new services.

Available services are written $a \Rightarrow P$, where a is the service name and P is its body. Notably, services are one-shot and not persistent by default: an invocation to a consumes the service definition. New services can be dynamically deployed, by using the action prefix $\mathsf{install}[a \Rightarrow P]$. This allows, combined with recursion, to program persistent services (see Example 2), for which we use the syntactic sugar

$$*a \Rightarrow P \tag{1}$$

A session is a logical unit of work, composed by different endpoints possibly distributed across sites. Each endpoint is written $r \rhd P_i$, where r is the session name and P_i is the i-th participant to session r.

Each service invocation prefix $\mathsf{invoke}\ a$ within a running session r has the effect of activating an instance $r \rhd P$ of the the service a (such as the one in (1)) that runs (inside the server context) as a new partner of session r. Basically, $r \rhd P$ is the service endpoint of session r.

Sessions can be nested at an arbitrary level of depth. Services are always installed at the top level and can be invoked from any level of nesting (the instance of the invoked service is opened, in the server context, as part of the innermost session containing the invoker).

Different endpoints $r \rhd P_1, ..., r \rhd P_n$ within the same session can communicate by message passing. The shared name r is used to guarantee that messages are exchanged only between partners of the same session, so that two instances $r_1 \rhd P$ and $r_2 \rhd P$ of the same persistent service $*a \Rightarrow P$ (that was invoked from two different sessions r_1 and r_2) run separately and cannot interfere.

The input prefix for intra-session communication is written $x(y)$, while intra-session output prefix is written $\overline{x}w$.

Sites. Since services are somewhat analogous to methods of a class, it is very frequent that a pool of services (and all their instances) must share some information (think e.g. of an airline reservation service, that must query and update the database of available seats). In μse this feature is realized by giving the possibility to group processes into sites and by adding communication primitives for intra-site communication. We write $l :: P$ where l is the site name (also called location) and P is the process located at l. Likewise sessions, sites are logical containers, not necessarily physical (machine-related) ones.

Intra-site input prefix is $x?(y)$ and intra-site output prefix is $x!w$. Likewise service invocation, intra-site communication is performed regardless of the session hierarchy.

Session merge. The most advanced feature of μse is the possibility of merging two separately running sessions. This is possible only if some of the endpoints of one session expose an entry point (prefix invite e, where e is the name of the entry point) and if some of the endpoints of the other session are willing to join (prefix accept e).

The invitation/acceptance is guided by an *entry point* which is a control flow mechanism for programming when processes can join sessions; for instance, using different entry points it is possible to let processes enter a session in different moments of the computation.

From the technical point of view the merging of the two sessions is realized by the explicit fusion of the two session names: from that point on, the two names can be used interchangeably everywhere needed.

2.2 μse syntax

We assume countable pairwise disjoint sets of names for

- communication channels $(x, y, \dots)$,
- services $(a, b, \dots)$,
- entry points $(e, f, \dots)$,
- sessions $(r, s, \dots)$ and
- locations $(l, \dots)$.

Channels, services and entry points can be exchanged in communications and are ranged over by $v, w, \dots$ while $n, m, \dots$ range over all names but locations.

$$
\begin{array}{rcll}
S, T & ::= & l :: a \Rightarrow P & \text{Service definition} \\
 & | & l :: P & \text{Located process} \\
 & | & S|T & \text{Composition of systems} \\
 & | & (\nu n)S & \text{New name} \\
 & & & \\
P, Q & ::= & \mathbf{0} & \text{Nil} \\
 & | & \overline{x}w.P & \text{Intra-session output} \\
 & | & x(y).P & \text{Intra-session input} \\
 & | & x!w.P & \text{Intra-site output} \\
 & | & x?(y).P & \text{Intra-site input} \\
 & | & \mathsf{install}[a \Rightarrow P].Q & \text{Service installation} \\
 & | & \mathsf{invoke}\ a.P & \text{Service invocation} \\
 & | & \mathsf{invite}\ e.P & \text{Entry point} \\
 & | & \mathsf{accept}\ e.P & \text{Acceptance of a new partner} \\
 & | & r \rhd P & \text{Session} \\
 & | & P|Q & \text{Parallel composition} \\
 & | & (\nu n)P & \text{New name} \\
 & | & \mathrm{rec}\ X.P & \text{Recursive process} \\
 & | & X & \text{Recursive call}
\end{array}
$$

Fig. 1. Syntax of processes

The syntax of $\mu\mathsf{se}$ is defined in Figure 1, where, in the last two productions of processes, variables (ranged over by $X, Y, \ldots$) account for the definition of recursive processes.

Systems (ranged over by $S, T, \ldots$) are parallel compositions of a finite number of *locations* where services are published and processes executed. A location where a service a is defined is meant to be the domain into which all instances of a are executed when a is invoked.

A $\mu\mathsf{se}$ process can be the empty process (we will drop trailing $\mathbf{0}$s), a process prefixed with an action (discussed in the following), a process running in a session, the parallel composition of processes, a process under a name restriction, a recursive process or a recursive invocation.

Processes (ranged over by $P, Q, \ldots$) communicate via channels very much like e.g., π-calculus processes, according to two featured modalities: *intra-session* and *intra-site* communication.

As outlined before, intra-session communications are used to let different endpoints of the same session to interact despite their running sites. Hence, processes located at different sites but sharing a session can interact via intra-session input and output. Conversely, intra-site communications allow different processes to communicate, provided that they are in the same site. This is used to model local communications and eases the programming of activities that are independent of the specific session. For instance, a program that counts the invocations to services defined at a given location can be programmed simply as a located process that increments a given variable when it receives a service

$$\mathcal{A}|\mathcal{A}' \equiv \mathcal{A}'|\mathcal{A} \qquad \mathcal{A}|\mathbf{0} \equiv \mathcal{A} \qquad (\mathcal{A}|\mathcal{A}')|\mathcal{A}'' \equiv \mathcal{A}|(\mathcal{A}'|\mathcal{A}'')$$
$$(\nu n)(\mathcal{A}|\mathcal{A}'') \equiv \mathcal{A}|(\nu n)\mathcal{A}'', \text{ if } n \notin \mathrm{fn}(\mathcal{A})$$
$$(\nu n)(\nu m)\mathcal{A} \equiv (\nu m)(\nu n)\mathcal{A} \qquad (\nu n)\mathcal{A} \equiv \mathcal{A}, \text{ if } n \notin \mathrm{fn}(\mathcal{A}) \qquad (\nu r)(r \doteq s) \equiv \mathbf{0}$$
$$r \doteq r \equiv \mathbf{0} \qquad \mathrm{rec}\ X.P = P\{\mathrm{rec}\ X.P/X\}$$
$$l :: P|l :: Q \equiv l :: (P|Q) \qquad l :: (\nu n)P \equiv (\nu n)(l :: P)$$
$$r \rhd (\nu n)P \equiv (\nu n)(r \rhd P), \text{ if } n \neq r$$
$$r \doteq s|P \equiv r \doteq s|P\{r/s\} \qquad r \doteq s \equiv s \doteq r$$
$$r \rhd (s \doteq t|P) \equiv s \doteq t|r \rhd P \qquad l :: (r \doteq s|P) \equiv r \doteq s|l :: P$$

Fig. 2. μse structural congruence

name via intra-site input on a given channel x; on invocation, each service sends its name to the counter with an intra-site output on x.

Processes can install new service definitions on their running locations. Service invocations enable processes to start new sessions with services; it is worth noting that an invoker does not need to know the location of the invoked service. Finally, a mechanism for letting processes to join existing sessions is given by the prefixes invite e and accept e.

Prefixes can be divided into two classes: *session-dependent* and *session-independent*. Intuitively, the former are those prefixes whose execution is dependent of and must be executed within a session; while session independent prefixes do not depend on sessions and can be executed also outside them. The distinction will be more clear when the operational semantics will be given, for the moment it suffices to say that intra-session input/output, invitation/acceptance and service invocation are session-dependent, while intra-site input/output are session-independent.

Finally, usual process algebraic operators like parallel composition and name restriction are introduced, the latter is one of the binders of μse. Indeed, the occurrences of y and n are bound in $a \Rightarrow P$, $x(y).P$, $x?(y).P$, $(\nu n)P$ and $(\nu n)S$. Therefore, the typical definition of sets of free, bound and all the names, respectively written as fn(_), bn(_) and n( ), are assumed for systems and processes; moreover, bound names can be safely alpha renamed.

2.3 μse operational semantics

The semantics of μse requires a structural congruence relation and an extended syntax, namely *explicit substitutions* $r \doteq s$ of sessions. (Explicit substitutions are session-independent.) Let $\mathcal{A}, \mathcal{B}$ to range over systems (including explicit substitutions) and processes.

$$\overline{x}w.P \xrightarrow{\overline{x}w} P \qquad x!w.P \xrightarrow{x!w} P$$

$$x(y).P \xrightarrow{xv} P\{v/y\} \qquad x?(y).P \xrightarrow{x?v} P\{v/y\}$$

$$\mathsf{invoke}\ a.P \xrightarrow{\perp a} P \qquad \mathsf{install}[a \Rightarrow R].P \xrightarrow{a[R]} P$$

$$\mathsf{invite}\ e.P \xrightarrow{\vdash e} P \qquad \mathsf{accept}\ e.P \xrightarrow{\dashv e} P$$

$$\frac{P \xrightarrow{\alpha} Q \quad \alpha \in \{\perp a, xv, \overline{x}v, \vdash e, \dashv e\}}{r \rhd P \xrightarrow{r\ \alpha} r \rhd Q} \qquad \frac{P \xrightarrow{\alpha} Q \quad \alpha \notin \{\perp a, xv, \overline{x}v, \vdash e, \dashv e\}}{r \rhd P \xrightarrow{\alpha} r \rhd Q}$$

$$\frac{P \xrightarrow{a[R]} Q}{l :: P \xrightarrow{\tau} l :: Q \,|\, l :: a \Rightarrow R} \qquad \frac{P \xrightarrow{\alpha} Q \quad \alpha \notin \{a[R], x?(v), x!v\}}{l :: P \xrightarrow{\alpha} l :: Q}$$

$$l :: a \Rightarrow P \xrightarrow{r \top a} l :: r \rhd P \qquad \frac{P \xrightarrow{x!v} P' \quad Q \xrightarrow{x?v} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}' \quad \mathrm{bn}(\alpha) \cap \mathrm{fn}(\mathcal{B}) = \emptyset}{\mathcal{A}|\mathcal{B} \xrightarrow{\alpha} \mathcal{A}'|\mathcal{B}} \qquad \frac{\mathcal{A} \xrightarrow{r\ \overline{x}v} \mathcal{A}' \quad \mathcal{B} \xrightarrow{r\ xv} \mathcal{B}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'}$$

$$\frac{\mathcal{A} \xrightarrow{r \vdash e} \mathcal{A}' \quad \mathcal{B} \xrightarrow{s \dashv e} \mathcal{B}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'|s \doteq r} \qquad \frac{S \xrightarrow{r \top a} S' \quad T \xrightarrow{r \perp a} T'}{S|T \xrightarrow{\tau} S'|T'}$$

$$\frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}' \quad n \notin \mathrm{n}(\alpha)}{(\nu n)\mathcal{A} \xrightarrow{\alpha} (\nu n)\mathcal{A}'} \qquad \frac{\mathcal{A} \xrightarrow{\alpha} \mathcal{A}' \quad \alpha \in \{\overline{x}w, x!w, r\ \overline{x}w, r\ x!w\}}{(\nu w)\mathcal{A} \xrightarrow{(w)\alpha} \mathcal{A}'}$$

Fig. 3. Operational semantics

Definition 1 (μse structural congruence). *The structural congruence over μse systems (and processes) is the smallest equivalence relation satisfying the axioms in Table 2.*

Structural congruence $\equiv$ includes associativity, commutativity and identity over nil for parallel composition and rules for scope extrusion. Also, $\equiv$ gives the semantics of $r \doteq s$ in terms of substitution. Notice that explicit substitutions can freely "float" in the term structure and are persistent, unless a restriction forbids movements.

Definition 2 (μse lts semantics). *The μse lts semantics is the least lts generated by the inference rules in Figure 3, closed under structural congruence.*

The rules for prefixes simply execute them, moving the information to the label. Notice that in the case of input prefixes the input value is guessed, as usual for early style semantics, and the substitution of the actual value for the input variable performed immediately. Sessions are transparent to most of the actions, while a session name is added to the label in case of session-dependent actions (intra-session communication, invoke, invite and accept). Notice that only the name of the innermost session is added. Install requests are executed when the outermost level of the site is reached. Notice that sites are transparent to all actions but install and intra-site communications. Service definitions can produce sessions, and the session name is guessed in the early style. Notice that most of the synchronization rules can be applied both at the process and at the system level. The only exceptions are intra-site communication, which is meaningful only at the process level, and service invocation, which can be stated only at the system level since definitions are always at top level and can thus be isolated using the structural congruence. Finally, restriction is usually dealt with by moving restrictions at top level using structural congruence, but the rule for extrusions is necessary for interactions with the environment (and notably for bisimulation).

3 μse examples

In this section we illustrate the main features of $\mu\mathsf{se}$ by means of some examples. First, Example 1 shows a trivial client-server application while Example 2 shows how persistent services can be programmed. The most complex examples are the last two; Example 3 yields a complex multi-user game scenario and Example 4 illustrates how a service proxy can be specified transparently to the client.

Example 1. A simple server provides an "increment" service to increment integers interacting with clients according to a request-response protocol. Client P_c and server P_s can be simply programmed as

$$P_c = \overline{data}\ v.ret(v').P, \qquad P_s = data(w).\overline{ret}\ w+1$$

i.e., P_c sends the value v to the server, waits a result, and then continues executing as P and, accordingly, P_s receives a value w and returns the successor of w. (For simplicity we assume here that arithmetical operators are also available in the $\mu\mathsf{se}$ calculus.)

The system below assumes a client that runs under the session r at the site l_1 and invokes a service inc, while the definition of the service inc is available at the site l_2. The $\mu\mathsf{se}$ term representing the system is below.

$$l_1 :: r \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_2 :: inc \Rightarrow P_s$$

The complete system evolves as follows:

$$\begin{array}{r}
l_1 :: r \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_2 :: inc \Rightarrow P_s \xrightarrow{\tau} \\
l_1 :: r \rhd P_c \quad | \quad l_2 :: r \rhd P_s \xrightarrow{\tau} \\
l_1 :: r \rhd ret(v').P \quad | \quad l_2 :: r \rhd \overline{ret}\ v+1 \xrightarrow{\tau} \\
l_1 :: r \rhd P\{v+1/v'\} \quad | \quad l_2 :: r \rhd \mathbf{0} \quad
\end{array}$$

In words, upon service invocation P_s is executed as a new endpoint of session r. The two protocols exploit intra-session communication until the service concludes its task and the client prosecutes its execution as $P\{v+1/v'\}$ within r.

The definition of the service inc in Example 1 is consumed as soon as it is invoked. The next example shows how persistent service definitions can be programmed using recursion.

Example 2. A persistent inc service can be defined as follows

$$l_2 :: inc \Rightarrow \text{rec } X.(P_s \mid \mathsf{install}[inc \Rightarrow X])$$

(which, by using the aforementioned notation, can be written as $l_2 :: *inc \Rightarrow P_s$).

We consider now a system involving the service inc and two clients running in separate sites (and in separate sessions) but executing analogous protocols:

$$l_0 :: r_0 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_1 :: r_1 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_2 :: *inc \Rightarrow P_s$$

The complete system evolves as follows:

$$\begin{array}{c}
l_0 :: r_0 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_1 :: r_1 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_2 :: *inc \Rightarrow P_s \xrightarrow{\tau} \\
l_0 :: r_0 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_1 :: r_1 \rhd P_c \quad | \quad l_2 :: r_1 \rhd P_s|\mathsf{install}[inc \Rightarrow \ldots] \xrightarrow{\tau} \\
l_0 :: r_0 \rhd \mathsf{invoke}\ inc.P_c \quad | \quad l_1 :: r_1 \rhd P_c \quad | \quad l_2 :: *inc \Rightarrow P_s|r_1 \rhd P_s \xrightarrow{\tau} \\
l_0 :: r_0 \rhd P_c \quad | \quad l_1 :: r_1 \rhd P_c \quad | \quad l_2 :: *inc \Rightarrow P_s|r_1 \rhd P_s|r_0 \rhd P_s
\end{array}$$

At the end there are two instances of the server protocol P_s running on the same site l_2, but under different sessions (i.e., r_0 and r_1), and two instances of the client protocol running on different sessions at different sites. We remark that the session mechanism of μse will distinguish the instances of channels $data$ and ret used by sessions r_0 and r_1, and will allow synchronizations only among channels belonging to the same session.

In the next example we illustrate how entry points can be used for modeling a distributed game scenario.

Example 3. Consider a server s that waits for the connection of two different players. Whenever two players connect to the service, they share a session so that they can play together. For simplicity, we let $P_p = start().P$ be the protocol that any player follows after invoking s. When s signals the start of the game on the channel $start$, the players run as P which codes the logic of the game.

The server has two different states that, when invoked, respectively generate an instance of the following protocols

$$G_1 = \mathsf{invite}\ e.\overline{start}, \qquad G_2 = \mathsf{accept}\ e.\overline{start}.$$

Intuitively, when G_1 runs in a session r, it invites players to join r over the entry point e, and then sends to the player the message $start$. G_2 accepts an invitation for joining the session over the entry point e and then it sends the message $start$ to the player.

The game service is $s \Rightarrow G$ where

$$G = \text{rec } X.(G_1 \mid \mathsf{install}[s \Rightarrow (G_2 \mid \mathsf{install}[s \Rightarrow X])])$$

notice that the changes of the state of the server are modeled by using the primitive $\mathsf{install}[\ldots]$ for installing a new definition of the service.

Let us consider the following system

$$l_0 :: r_0 \rhd \mathsf{invoke}\ s.P_p \quad | \quad l_1 :: r_1 \rhd \mathsf{invoke}\ s.P_p \quad | \quad l_g :: s \Rightarrow G$$

composed by two players $T_0 = l_0 :: r_0 \rhd \mathsf{invoke}\ s.P_p$ and $T_1 = l_1 :: r_1 \rhd \mathsf{invoke}\ s.P_p$ together with a game server $T = l_g :: s \Rightarrow G$.

In the first reduction, one of the players, say T_1 invokes the server

$$\begin{array}{r} T_0 \quad | \quad T_1 \quad | \quad T \xrightarrow{\tau} \\ T_0 \quad | \quad l_1 :: r_1 \rhd P_p \quad | \quad l_g :: r_1 \rhd G \end{array}$$

Then the first install "redefines" s by installing it in the state G_2

$$\begin{array}{r} T_0 \quad | \quad l_1 :: r_1 \rhd P_p \quad | \quad l_g :: r_1 \rhd G \xrightarrow{\tau} \\ T_0 \quad | \quad l_1 :: r_1 \rhd P_p \quad | \quad l_g :: r_1 \rhd G_1 \mid l_g :: s \Rightarrow (G_2 \mid \mathsf{install}[s \Rightarrow G]) \equiv \\ l_1 :: r_1 \rhd P_p \quad | \quad T_0 \quad | \quad l_g :: s \Rightarrow (G_2 \mid \mathsf{install}[s \Rightarrow G]) \mid r_1 \rhd G_1 \end{array}$$

Now player T_0 can invoke s so that the server (in state G_2) is executed.

$$\begin{array}{r} l_1 :: r_1 \rhd P_p \quad | \quad T_0 \quad | \quad l_g :: s \Rightarrow (G_2 \mid \mathsf{install}[s \Rightarrow G]) \mid r_1 \rhd G_1 \xrightarrow{\tau} \\ l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: r_0 \rhd (G_2 \mid \mathsf{install}[s \Rightarrow G]) \mid r_1 \rhd G_1 \end{array}$$

observe that G_2 and T_0 are in a session different from G_1 and T_1; moreover, the players do not know which state s is in when they invoke it.

By executing the second install, the service becomes again available, in the initial state.

$$\begin{array}{r} l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: r_0 \rhd (G_2 \mid \mathsf{install}[s \Rightarrow G]) \mid r_1 \rhd G_1 \xrightarrow{\tau} \\ l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: r_0 \rhd G_2 \mid r_1 \rhd G_1 \mid l_g :: s \Rightarrow G \equiv \\ l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: s \Rightarrow G \mid r_0 \rhd G_2 \mid r_1 \rhd G_1 \end{array}$$

Finally, G_1 and G_2 can synchronize so that, after their sessions are coalesced, they will signal to the players that the game can start.

$$\begin{array}{r} l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: s \Rightarrow G \mid r_0 \rhd G_2 \mid r_1 \rhd G_1 \xrightarrow{\tau} \\ l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: s \Rightarrow G \mid r_0 \rhd \overline{start} \mid r_1 \rhd \overline{start} \mid r_0 \doteq r_1 \end{array}$$

At this point both players are running on the same session since $r_0 \doteq r_1$, and after receiving the message $\overline{start}$ they may start running the protocol P_p. Moreover, the server is ready for accepting a new request. The very last transitions let players start the game.

$$\begin{array}{r} l_1 :: r_1 \rhd P_p \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_g :: s \Rightarrow G \mid r_0 \rhd \overline{start} \mid r_1 \rhd \overline{start} \mid r_0 \doteq r_1 \xrightarrow{\tau}{}^{*} \\ l_1 :: r_0 \rhd P \quad | \quad l_0 :: r_0 \rhd P \quad | \quad l_g :: r_0 \rhd \mathbf{0} \mid r_1 \rhd \mathbf{0} \mid s \Rightarrow G \mid r_0 \doteq r_1 \end{array}$$

Note that the entry point e could be generated fresh at each execution to guarantee that two players that consecutively invoke s will always play together.

$$G = \text{rec } X.(\nu e)(G_1 \,|\, \mathsf{install}[s \Rightarrow (G_2 \,|\, \mathsf{install}[s \Rightarrow X])])$$

Next example shows how proxies can be easily programmed and exploits the intra-site communication of $\mu\mathsf{se}$.

Example 4. Consider a set of different services $s_0, \ldots, s_n$ providing the same functionality P, any of them running on a different site. Let us assume the services to be persistent and defined as

$$S_i = l_i :: *s_i \Rightarrow P$$

Moreover, we assume that each client wants to access the services in a transparent way, i.e., by invoking a service s that acts as a proxy, and forwards the invocation to one of the actual services.

As a first solution, we can model the proxy as a service that non-deterministically selects one of the available providers, as below

$$Ps = \prod_i Av_i \quad | \quad * \, s \Rightarrow av?(x).\mathsf{invoke}\ x$$

Any process $Av_i = \text{rec } X.av!s_i.X$ gives a persistent witness of the fact that the service s_i is one of the available providers (in more complex situations, the description of available services may take load balancing into account, exploiting e.g. a sequential list of invocations). The actual definition of the proxy $*s \Rightarrow av?(x).\mathsf{invoke}\ x$ states that once the proxy is invoked it uses intra-site communication to select one of the available services, and then invokes it. If we consider a client that invokes s and then behaves like Q, the whole system behaves as follows.

$$\prod_i S_i \quad | \quad l_p :: Ps \quad | \quad l_c :: r \rhd \mathsf{invoke}\ s.Q \xrightarrow{\tau}$$
$$\prod_i S_i \quad | \quad l_p :: Ps \,|\, r \rhd av?(x).\mathsf{invoke}\ x \quad | \quad l_c :: r \rhd Q \xrightarrow{\tau}$$
$$\prod_i S_i \quad | \quad l_p :: Ps \,|\, r \rhd \mathsf{invoke}\ s_k \quad | \quad l_c :: r \rhd Q \xrightarrow{\tau}$$
$$\prod_{i \neq k} S_i \,|\, l_k :: r \rhd P \,|\, * \, s_k \Rightarrow P \quad | \quad l_p :: Ps \,|\, r \rhd \mathbf{0} \quad | \quad l_c :: r \rhd Q$$

Note that from this moment on the client at site l_c and the activated instance of the selected service at site l_k will share the same session r.

4 Observational semantics of $\mu\mathsf{se}$

This section proposes an observational semantics of $\mu\mathsf{se}$ relying on the well-known notion of bisimulation. We prefer to use weak bisimulation as it is more suitable for reasoning on μse systems and, more generally, for giving coarser equivalence relations amongst systems.

Let $\Rightarrow$ be the reflexive and transitive closure of $\xrightarrow{\tau}$. Let us denote relation composition as juxtaposition (e.g., $\Rightarrow\xrightarrow{\alpha}$ is the composition of relations $\Rightarrow$ and $\xrightarrow{\alpha}$). Let $\stackrel{\hat{\alpha}}{\Rightarrow}$ be $\Rightarrow\xrightarrow{\alpha}\Rightarrow$ if $\alpha \neq \tau$ and $\Rightarrow$ if $\alpha = \tau$.

Definition 3 (Bisimilarity). *A binary relation $\mathcal{B}$ on systems is a* (weak) μse bisimulation *if it is symmetric and for any $(S,T) \in \mathcal{B}$*

- *for each $S \xrightarrow{\alpha} S'$ such that* $\mathrm{bn}(\alpha) \cap \mathrm{fn}(T) = \emptyset$ *we have $T \stackrel{\hat{\alpha}}{\Rightarrow} T'$ with $(S',T') \in \mathcal{B}$;*

Bisimilarity is the largest bisimulation.

We will show here that bisimilarity can be used to analyze properties of services, in particular to show that an implementation of a service is compliant (i.e., bisimilar) to a simpler specification.

Let us consider a simple service that computes some mathematical function *fun* (such as the increment in Example 1, or even better some computationally expensive function). We can write the specification as:

$$l :: *g \Rightarrow P \qquad \text{with } P = data(x).\overline{ret}\ fun(x)$$

The only possible transitions for this service are acceptance of invocations at g, followed by a protocol in the created session composed by an input on *data* and an output on *ret*. Notice that many instances can be executed at the same time, but they will not interfere since they run all on different sessions.

Following the ideas in Example 4, a first implementation could simply ask another service s_i from a given pool $s_1, \ldots, s_n$ to do the job, choosing it nondeterministically.

$$l :: (\nu s_1 \ldots s_n)((\nu av)(\prod_{i=1,\ldots,n} \mathrm{rec}\ X.av!s_i.X \mid *\, g \Rightarrow av?(x).\mathsf{invoke}\ x)| \\ \prod_{i=1,\ldots,n} *s_i \Rightarrow P_i)$$

In the refined implementation the invocation is mimicked by the invocation of g, a communication on av (and since av is restricted no other transitions are possible) and the invocation of the chosen service s_i (which will then be executed inside the same session). Notice that these two last transitions are just τ. If all the bodies of services s_i are bisimilar to P, then the interaction is executed as in the previous case. Also, removing e.g. the restriction on av breaks the bisimilarity, since the implementation of g could then interact with another channel av in the environment, while the specification does not allow this interaction.

In another possible implementation, g can invite other sessions to do the job. For simplicity we consider just one such session, but the case of a nondeterministic choice among many equivalent sessions is analogous.

$$l :: (\nu e) * g \Rightarrow \mathsf{invite}\ e\ |\mathrm{rec}\ X.(\nu r) r \rhd \mathsf{accept}\ e.(P|X)$$

In this case the invocation in the specification is simulated by the invocation in the implementation plus the invite. Notice that e should be bound to avoid other sessions to come into play instead of the wanted one. After the invitation, the session created by g and r (which is restricted to avoid different recursive calls to interfere) are merged. Notice that other instances create further sessions, but this is unimportant since only the most internal one matters, e.g., $r \rhd r' \rhd P$ is bisimilar to $r' \rhd P$.

5 Concluding remarks and future work

We have introduced μse, a process calculus tailored to handle multiparty sessions in service oriented scenarios. The presentation includes the full formalization of the operational semantics in the SOS style and the definition of a bisimilarity-based abstract semantics.

μse collects ideas emerged in recent work, but has several degrees of originality. The main sources of inspiration have been the π-calculus [11] (how could it be otherwise when communication is involved!) and SCC [1, 8], a proposal based on two-party, bidirectional sessions. Yet μse and SCC differ on many design choices. For example, the invocation of a service always opens a new session in SCC both on the client and on the service side. In μse instead joining sessions can be both freshly generated or already running. Another difference is that SCC offers more constrained forms of local communication: pipelining [1] and data streaming [8]. In the case of μse it should be possible to simulate both such communication styles using intra-site communication over dedicated channels.

As argued in the introduction μse is also very different from correlation-based calculi like SOCK [3] and COWS [9]. The global calculus [4] allows for multi-party sessions, but, roughly speaking, session identifiers are modeled just as pi-calculus channel names (freshly created and distributed to participants during the initialization phase of the service protocol). In μse instead sessions offer a logical context for driving communication on top of intra-session channel names. Moreover, entry points allow to dynamically merge running sessions, an operation not possible in the global calculus.

Note also that μse is different from ambient-inspired calculi [5]: as a matter of fact, sessions are not ambients, because they can neither be opened nor moved. Moreover, intra-session communication is very different from the kind of intra- and inter-ambient communications considered in the various flavors of ambient calculus.

We conclude by enumerating some of the possible extensions of μse that have been already considered in the design phase of μse primitives.

The first issue regards the nesting of sessions. In the current version nesting is mainly used to forbid the use of intra-session communication within the same party. For example P and Q can neither carry an intra-session interaction in $r \rhd (P|Q)$ nor in $r \rhd (P|r \rhd Q)$. Thus we dislike a flattening structural congruence rule such as $r \triangleright (P|s \triangleright Q) \equiv r \triangleright P|s \triangleright Q$ and prefer to maintain the logical structure of the computation through session nesting. More importantly, session nesting can play a prominent role when explicit session closure primitives are introduced (like those available in SCC and COWS). For instance, sessions can confine the effect of closure mechanisms so that determining which part of a running session must be terminated is straightforward.

The second issue concerns the kind of communication exploited within sessions. In fact, while CCS-like communication [10] is the obvious choice when only two-party sessions are considered, in the presence of multi-party sessions a more natural (but sophisticated) alternative would be some variant of multicast (like broadcast [6] or CSP-like interaction [7], or even some combination of different policies [2]).

References

1. 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 M. Bravetti, M. Núñez, and G. Zavattaro, editors, *Proceedings of WS-FM 2006*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer Verlag, 2006.
2. R. Bruni and I. Lanese. PRISMA: A mobile calculus with parametric synchronization. In U. Montanari, D. Sannella, and R. Bruni, editors, *Proceedings of TGC'06*, volume 4661 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2007. To appear.
3. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In P. Ciancarini and H. Wiklicky, editors, *Proceedings of COORDINATION'06*, volume 4038 of *Lect. Notes in Comput. Sci.*, pages 63–81. Springer Verlag, 2006.
4. M. Carbone, K. Honda, N. Yoshida, and R. Milner. Structured communication-centred programming for web serices. In R. De Nicola, editor, *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2007.
5. L. Cardelli and A. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.
6. C. Ene and T. Muntean. A broadcast-based calculus for communicating systems. In *Proceedings of IPDPS'01*. IEEE Computer Society, 2001.
7. C. Hoare. A model for communicating sequential processes. In R. McKeag and A. Macnaghten, editors, *On the Construction of Programs*. Cambridge University Press, 1980.
8. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proceedings of SEFM'07*, 2007. To appear.
9. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In R. De Nicola, editor, *Proceedings of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 33–47. Springer Verlag, 2007.
10. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer Verlag, 1980.
11. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.

12. OASIS. *UDDI - Universal Description, Discovery and Integration of Web Services.* http://www.uddi.org/specification.html.
13. OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft.* http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf.
14. Sensoria Project. Public web site. http://sensoria.fast.de/.
15. World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework.* http://www.w3.org/TR/soap12-part1/.
16. World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0. Working draft 17 December 2004.* http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/.
17. World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1.* http://www.w3.org/TR/wsdl.