Multiparty sessions in SOC*

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

[1] Dipartimento di Informatica, Università di Pisa, Italy
bruni@di.unipi.it
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy
lanese@cs.unibo.it
[3] Departamento de Computación, Universidad de Buenos Aires, Argentina
hmelgra@dc.uba.ar
[4] Department of Computer Science, University of Leicester, UK
et52@mcs.le.ac.uk

Abstract. Service oriented applications feature interactions among several 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) envisages systems as a combination of *services*, possibly provided by different organizations. Typically, a service can be concurrently requested by many invokers (e.g., users or other services) so that many service instances can be carried on at the same time (e.g., several customers booking flights from the same airline). Hence, it is important to guarantee that the interactions taking place in different instances do not interfere, and messages are routed to the intended recipients.

Emerging standards like WS-BPEL [23] and WS-CDL [25] exploit the idea of *correlation sets*, which allow messages to be routed to specific instances of services depending on a pre-defined subset of the invocation parameters (e.g., requests are routed according to usernames). Though correlation sets guarantee a good expressiveness, we argue that they make analysis harder because the emerging patterns of interaction rely on data values. Also, unrelated sessions can interfere with each other if they know (or use by chance) the “right” values.

Some formal methods [3, 19, 2, 16, 4, 12] advocate the concept of *session* as an abstraction mechanism for enclosing an arbitrarily complex interaction between

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

two partners in order to guarantee e.g., that, during a conversation, messages are routed as desired. As observed e.g. in [2], since modern distributed systems rely on the TCP/IP stack, it is usually accepted that sessions involve only two participants (usually according to the client/server architecture). Consider a scenario where a customer c of a bank b wants to withdraw some money from an ATM a; this can be modeled as c invoking the service a that in turn invokes b, which closes the protocol by e.g., sending a text message on c's mobile phone. Usually, two different sessions, say s and s', are used during the computation (see, e.g., [2]): s is a session between c and a while s' between a and b. Typically, this forces the programmer to explicitly handle communication of s and s' in order to relate events occurring in the two sessions.

In this paper we propose $\mu\mathsf{se}$ (read "muse", after "MUltiparty SEssion"), a process calculus whose primitives are designed for easing the programming of SOC systems using *multiparty sessions*, namely sessions to which more than two actors can take part, as a high-level abstraction mechanism to coordinate interactions among several participants. We also intend to highlight the relations between sessions and the other main features of SOC: services, communication protocols, sites, etc.

One of $\mu\mathsf{se}$ main design principles is that programmers should be relieved from the explicit handling of session identifiers. The rationale being that, in our opinion, SOC systems should be programmed by abstracting from the error-prone activity of session handling. Rather, the language should support the implicit creation and exchange of session identifiers. For instance, the previous ATM scenario can be more easily programmed in $\mu\mathsf{se}$ by merging s and s' into a session delimiting exactly c, a and b; $\mu\mathsf{se}$ semantics then guarantees that interactions among c, a and b are not disturbed by external processes (cf. Example 3). Another example can be an online game where a server provides the playing platform and users can log into different games. While logged in, users "in the same room" can interact according to a game-specific protocol without interference from users in other rooms. This can be naturally modeled using a multiparty session for each room that would avoid the complication of maintaining the association room/players required if two-party sessions (each containing player and server) were used.

Another important design choice concerns $\mu\mathsf{se}$ communications, which can be *intra-* or *extra-session*. More precisely, $\mu\mathsf{se}$ requires that participants on different sites always use intra-session communications, namely they must be endpoints of the same session. Instead, processes at the same site are allowed to communicate also across sessions. Intuitively, co-located processes can exploit local resources (e.g., databases, file systems, etc.) to interact, while remote processes must rely on underlying middlewares (like TCP/IP, SOAP, ...).

To this end, $\mu\mathsf{se}$ systems consist of sites where services, sessions and processes live. Services can be dynamically published, sessions are dynamically created, new participants can join them at runtime and concurrent ongoing sessions can be merged. We equip $\mu\mathsf{se}$ with a (weak) bisimilarity-based equivalence whose appropriateness is illustrated by means of a small proxy scenario.

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

2 The μse calculus

This section introduces the μse calculus and its main features: (*i*) nested multiparty 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 installing 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 its service definition. New services are dynamically deployed using the prefix $\mathsf{install}[a \Rightarrow P]$, that, combined with recursion, permits to program persistent services (see Example 2 in Section 3), 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 $r \rhd Q$, where r is the session name and Q is one of the participants to session r. When the endpoint $r \rhd Q$ invokes a (executing prefix $\mathsf{invoke}\ a$), a new instance $r \rhd P$ of service a in (1) is activated on the service site as a new partner of session r. In fact, $r \rhd P$ is a 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 an endpoint of the innermost session containing the invoker.

Different endpoints $r \rhd P_1$, ..., $r \rhd P_n$ within the same session can interact by means of *intra-session* input and output prefixes (respectively written $x(y)$ and

$\overline{x}w$, reminiscent of π-calculus prefixes). The shared name r is used to guarantee that messages are exchanged only between partners of the same session. Hence, two instances $r_1 \rhd P$ and $r_2 \rhd P$ of (1), invoked from two endpoints of two different sessions r_1 and r_2, run separately and cannot interfere (unless r_1 and r_2 are merged).

Sites. We envisage services as somehow analogous to methods of a class; therefore, a pool of services (and all their instances) may share some information (e.g., the instances of an airline reservation service must query and update a flights database). In μse this feature is realized by giving the possibility to group processes into sites and by adding 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 and output prefix are respectively written $x?(y)$ and $x!w$ and, as service invocation, they are executed regardless of the session hierarchy. We could have also used local variables for intra-site communications, but we preferred message passing to follow the style of intra-session communications.

Merging sessions. The most advanced feature of μse is the possibility of merging two distinct running sessions. This is possible only when two endpoints expose the same *entry point* e via prefix $\mathsf{merge}^p\ e$, requiring to merge the respective sessions. Merge prefixes $\mathsf{merge}^p\ e$ are polarized with $p \in \{+, -\}$, with the obvious meaning that complementary merge actions on the same entry point e (i.e., $\mathsf{merge}^+\ e$ and $\mathsf{merge}^-\ e$) can synchronize.

Merging of sessions is guided by entry points e which yield 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 at different stages of the computation.

Technically, the merging of sessions is realized by explicit fusion of session names: after their fusion, two names can be used interchangeably wherever needed.

2.2 μse syntax

We assume that countable pairwise disjoint sets of names are available for

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

Channels, services and entry points are *communicable values* (which are ranged over by $v, w, \dots$) while sessions and locations cannot be communicated. We let $n, m, \dots$ range over all names but locations.

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

Fig. 1. Syntax of systems and processes

The syntax of μse is defined in Figure 1, where the last two productions for processes account for recursion ($X, Y, \ldots$ stand for process variables; we assume variables guarded by prefixes in the body of recursive definitions).

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 upon invocation.

A μse process can be the empty process (we will drop trailing $\mathbf{0}$s), a process prefixed by an action (discussed in the following), a process running in a session (endpoint), 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 regardless their running sites. Hence, processes located at different sites but sharing the same session can interact via intra-session input and output. Conversely, intra-site communications allow different endpoints to communicate, provided that they are running 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 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.

$$\mathcal{A}|\mathcal{A}' \equiv \mathcal{A}'|\mathcal{A} \qquad \mathcal{A}|\mathbf{0} \equiv \mathcal{A} \qquad (\mathcal{A}|\mathcal{A}')|\mathcal{A}'' \equiv \mathcal{A}|(\mathcal{A}'|\mathcal{A}'')$$

$$(\nu n)(\mathcal{A}|\mathcal{A}'') \equiv \mathcal{A}|(\nu n)\mathcal{A}'', \text{ if } n \notin \text{fn}(\mathcal{A})$$

$$(\nu n)(\nu m)\mathcal{A} \equiv (\nu m)(\nu n)\mathcal{A} \qquad (\nu n)\mathcal{A} \equiv \mathcal{A}, \text{ if } n \notin \text{fn}(\mathcal{A})$$

$$l :: P|l :: Q \equiv l :: (P|Q) \qquad l :: (\nu n)P \equiv (\nu n)(l :: P)$$

$$r \rhd (\nu n)P \equiv (\nu n)(r \rhd P), \text{ if } n \neq r$$

$$r \doteq r \equiv \mathbf{0} \qquad (\nu r)(r \doteq s) \equiv \mathbf{0} \qquad r \doteq s|P \equiv r \doteq s|P\{r/s\} \qquad r \doteq s \equiv s \doteq r$$

$$r \rhd (s \doteq t|P) \equiv s \doteq t|r \rhd P \qquad l :: (r \doteq s|P) \equiv r \doteq s|l :: P$$

$$\text{rec } X.P \equiv P\{\text{rec } X.P/X\}$$

Table 1. μse structural congruence

Processes can install new service definitions in their running locations. Service invocations enable processes to activate new endpoints on the service location. Note that service invocation requires only the service name, not its location, thus if many services with the same name are available one of them is chosen nondeterministically. Finally, a mechanism for letting processes join existing sessions is given by the prefix $\mathsf{merge}^p\ e$.

Prefixes can be divided into two classes: *session-dependent* and *session-independent*. Intuitively, the former are those whose execution is dependent of and can be executed only within a session; while session independent prefixes do not depend on sessions and can be executed also outside them. The distinction will be clearer when the operational semantics is given, for the moment it suffices to say that intra-session input/output, session merge 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. In fact, the occurrences of y and n are bound in $x(y).P$, $x?(y).P$, $(\nu n)P$ and $(\nu n)S$ and the typical definitions of set of free, bound and all names, respectively written as $\text{fn}(\_)$, $\text{bn}(\_)$ and $\text{n}(\_)$, are assumed for systems and processes. As usual, 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. Note that explicit substitutions are session-independent. Let $\mathcal{A}, \mathcal{B}$ range over systems (including explicit substitutions) and processes.

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

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

$$l :: a \Rightarrow P \xrightarrow{r\top a} l :: r \rhd P \qquad \mathsf{invoke}\ a.P \xrightarrow{\perp a} P \qquad \mathsf{install}[a \Rightarrow R].P \xrightarrow{a[R]} P$$

$$\mathsf{merge}^p\ e.P \xrightarrow{e^p} P$$

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

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

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

$$\frac{\mathcal{A} \xrightarrow{re^+} \mathcal{A}' \quad \mathcal{B} \xrightarrow{se^-} \mathcal{B}'}{\mathcal{A}|\mathcal{B} \xrightarrow{\tau} \mathcal{A}'|\mathcal{B}'|s \doteq r} \qquad \frac{S \xrightarrow{r\top a} S' \quad T \xrightarrow{r\perp a} T'}{S|T \xrightarrow{\tau} S'|T'}$$

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

Table 2. 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 1 (where* $\mathrm{fn}(\mathcal{A})$ *is defined as expected).*

Structural congruence $\equiv$ includes associativity, commutativity and identity over $\mathbf{0}$ for parallel composition and rules for scope extrusion. Also, $\equiv$ gives the semantics of recursion and $r \doteq s$ in terms of substitutions. Notice that any explicit substitution $r \doteq s$ is persistent and can freely "float" in the term structure, unless a restriction on r or s forbids its movements.

The operational semantics of μse is specified through a labeled transition system (lts). We use α to range over labels. Bound variables occurring in labels are in round parentheses, and functions $\mathrm{fn}(\_)$, $\mathrm{bn}(\_)$ and $\mathrm{n}(\_)$ are extended in the natural way to labels.

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

The rules for prefixes simply execute them, moving the information to the transition label. As usual for early semantics, input prefixes guess the actual value and immediately substitute it for the formal variable. 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 communications, invoke and merge). Notice that only the name of the innermost session is added. Service definitions can produce sessions, and the session name is guessed in the early style. Install requests are executed when the outermost level of the site is reached. Observe that sites are transparent to all actions but install and intra-site communications. Also, most of the synchronization rules can be applied both at the process and at the system level. The only exceptions are (*i*) intra-site communication, which is meaningful only at the process level, and (*ii*) service invocation, which can be stated only at the system level since definitions are always at the top level. Finally, restriction is dealt with by moving restrictions to the outermost level using structural congruence, but the rule for extrusions is necessary for interactions with the environment (and notably for bisimulation).

3 Programming in μse

This section illustrates the main features of μse by showing how it can be used to program simple SOC applications. Example 1 introduces a trivial client-server application, while Example 2 shows how persistent services can be programmed. To illustrate how multiparty sessions can be easily used, more complex scenarios are presented: Example 3 shows how a multiparty session can control interactions among three participants; Example 4 models a multi-player game; and Example 5 gives the definition of a proxy, which is transparent to clients.

Example 1. Consider the simple system below

$$l_c :: r \triangleright \mathsf{invoke}\ inc.P_c \quad | \quad l_s :: inc \Rightarrow P_s \tag{2}$$

where the client running at l_c in a session r invokes a service for incrementing integers on another location; client and service adopt a request-response protocol according to $P_c = \overline{data}\ v.ret(v').P$ and $P_s = data(w).\overline{ret}\ w{+}1$. Namely, P_c sends the value v to the service, waits a result, and then continues executing as P; accordingly, P_s receives a value w and returns the successor of w (arithmetical operators are assumed and they have precedence over other operators.)

The system (2) evolves as follows:

$$\begin{array}{rcll} l_c :: r \triangleright \mathsf{invoke}\ inc.P_c & | & l_s :: inc \Rightarrow P_s & \xrightarrow{\tau} \\ l_c :: r \triangleright P_c & | & l_s :: r \triangleright P_s & \xrightarrow{\tau} \\ l_c :: r \triangleright ret(v').P & | & l_s :: r \triangleright \overline{ret}\ v+1 & \xrightarrow{\tau} \\ l_c :: r \triangleright P\{v+1/v'\} & | & l_s :: r \triangleright \mathbf{0} & \end{array}$$

In words, upon service invocation, P_s is executed as a new endpoint of session r where intra-session communications let parameters to be passed around. $\square$

Observe that neither the client nor the service of Example 1 deal with session identifiers. Also, the definition of *inc* is consumed as soon as it is invoked. Nevertheless, persistent service definitions can be programmed by using recursion, as shown in the following example.

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

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

(which, by using the notation in (1), can be written as $l_s :: {*inc} \Rightarrow P_s$).

We consider now the case of two clients running in separate sessions (and in separate sites) but executing analogous protocols:

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

The complete system may reduce (in several steps) to

$$l_0 :: r_0 \rhd P_c \quad | \quad l_1 :: r_1 \rhd P_c \quad | \quad l_s :: {*inc} \Rightarrow P_s \quad | \quad l_s :: (r_1 \rhd P_s | r_0 \rhd P_s)$$

where two instances of the service protocol P_s run on l_s, but under different sessions r_0 and r_1, while two instances of the client run on different sessions at different sites. We remark that the session mechanism of $\mu\mathsf{se}$ will distinguish the instances of channels *data* and *ret* used by sessions r_0 and r_1, and will allow synchronizations only over channels belonging to the same session. □

Example 3. The ATM scenario described in Section 1 is shown. Consider

$$hiw :: r \rhd C \mid (\nu\ check, abort)(hiw :: {*atm} \Rightarrow A \mid branch :: {*bank} \Rightarrow B) \quad (3)$$

where C, A and B are respectively the customer, ATM and bank code (defined below); *check* and *abort* are private channels shared between A and B. For simplicity, we assume to have basic types (as numerals or strings), tuples (in angle brackets), nondeterministic choice, **if** statement and polyadic inputs (though channels are not typed). We enclose output tuples in angle brackets. The definition of C, A and B is as follows:

$$C = \mathsf{invoke}\ atm.\overline{req}\langle \mathtt{c}, \mathtt{m}\rangle.(cash(x)|sms(y).display!y)$$

$$A = req(x,y).\mathsf{invoke}\ bank.\overline{check}\langle x, y\rangle.(\ checked().\overline{cash}\ y + abort().\overline{cash}\ 0\ )$$

$$B = check(x,y).\mathbf{if}\ ok(x,y)\ \mathbf{then}\ \overline{checked}.\overline{sms}\ \mathtt{ok}\ \mathbf{else}\ \overline{abort}.\overline{sms}\ \mathtt{ko}$$

After invoking the ATM, C requests to withdraw an amount of money m offering some credentials c and waits for money and for an SMS confirmation. After invoking B, A forwards the request to B and waits for B's response either to confirm or abort the transaction (in which case no money is dispensed).

If the customer's credentials are valid, B confirms to A to proceed and notifies C by sending the ok SMS, which is diplayed on C's site. Observe that B enters the session between A and C after the latter invokes the bank service, hence the further interactions with C and A will not be messed up with possible concurrent sessions of the bank service. If the customer's credentials are invalid the bank let the ATM abort and sends a failure notification to the customer. □

An interesting aspect to highlight in Example 3 is the fact that the interactions between C and A or C and B are mediated by public channels and communications are hiddenly driven by sessions. More precisely, *req*, *cash* and *sms* can be thought of as the known ports through which participants communicate, and sessions avoid interferences among possible concurrent invocations of the ATM and bank services. Also, notice that exactly the same definition of the bank service can be installed on other locations so modeling the existence of different branches without affecting the customer.

In the next example we illustrate how entry points can be used for modeling a distributed game scenario where the number of participants is unbounded.

Example 4. Let s be a service that waits for the connection of *at least two* players. Whenever two players connect to the service, they share a session and a match starts. New participants may later join. For simplicity, we let $P_p = start().P$ be the protocol that any player follows after invoking s. When s signals the beginning of the match on the channel *start*, the players run as P, which codes the (unspecified) logic of the game.

The service has two different states that respectively generate an instance of the following protocols

$$G_1 = \mathsf{merge}^- \ e.\overline{start}.\mathrm{rec}\ X.\mathsf{merge}^- \ e.X, \qquad G_2 = \mathsf{merge}^+ \ e.\overline{start}$$

Intuitively, G_1 stands for the protocol followed by s for handling the first connection. Note that G_1 will run in a session, say r, and it will wait a player to join r over the entry point e. After the second player arrives, it sends to the player the message *start* and will repeatedly wait for new arrivals. By contrast, G_2 manages all subsequent invocations. In particular, G_2 joins an existing session over the entry point e and then it sends the message *start* to the player.

The game service is $s \Rightarrow G$ where

$$G = G_1 \,|\, \mathsf{install}[*s \Rightarrow G_2]$$

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

Let us consider the following system

$$l_0 :: r_0 \rhd \mathsf{invoke}\ s.P_p \quad | \quad l_1 :: r_1 \rhd \mathsf{invoke}\ s.P_p \quad | \quad l_g :: s \Rightarrow G \tag{4}$$

composed by two players and a game service. After several steps, system (4) may evolve to

$$l_0 :: r_0 \rhd P_p \quad | \quad l_1 :: r_1 \rhd P_p \quad | \quad l_g :: *s \Rightarrow G_2 \quad | \quad l_g :: (r_0 \rhd G_1 \,|\, r_1 \rhd G_2)$$

and G_1 and G_2 can finally synchronize (over the entry point e) so that, after their sessions are coalesced, they signal to the two players that the game starts.

$$\begin{gathered} r_0 \doteq r_1 \quad | \quad l_0 :: r_0 \rhd P_p \quad | \quad l_1 :: r_0 \rhd P_p \quad | \\ l_g :: *s \Rightarrow G_2 \quad | \quad l_g :: (r_0 \rhd \overline{start}.\mathrm{rec}\ X.\mathsf{merge}^- \ e.X \,|\, r_0 \rhd \overline{start}) \end{gathered}$$

Note that new invocations of s will create service sessions of the form $r \rhd G_2$. These sessions will join the first created session r_0, by merging over the entry point e. $\square$

The gaming example serves to show a nice feature of μse: players protocol need not to be aware of the order in which connections are established, i.e. any player can invoke the gaming service regardless of the fact that a session has been already started or not. Clearly, more complex game services may be written; for instance, a service that allows only a bounded number of participants and creates a new instance of the game when the bound is reached can be simply implemented using freshly created entry points. Remarkably, the programming of the counting mechanism can be straightforwardly achieved by using a local shared counter and intra-site communications.

Our last example shows how proxies can be easily programmed and exploits the intra-site communication of μse.

Example 5. Consider a set of different services $s_0, \dots, s_n$ 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 nondeterministically selects one of the available providers, as below

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

Any process $Av_i = \mathrm{rec}\ X.av!s_i.X$ 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 continues like Q, the whole system behaves as follows.

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

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

4 Observational semantics of μse

This section proposes an observational semantics of μse 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 $\stackrel{\alpha}{\Rightarrow}$ 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 $\mathrm{bn}(\alpha) \cap \mathrm{fn}(T) = \emptyset$, $T \stackrel{\alpha}{\Rightarrow} T'$ with $(S', T') \in \mathcal{B}$.*

Bisimilarity is the largest bisimulation.

Definition 3 instantiates the standard notion of weak bisimilarity for μse.

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

Let us consider a simple service a 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:

$$l :: *a \Rightarrow P \qquad \text{with } P = \mathit{data}(x).\overline{\mathit{ret}}\ \mathit{fun}(x) \tag{5}$$

The only possible transitions for this service are acceptance of invocations at a, followed by a protocol in the created session composed by an input on *data* and an output on *ret*.

Following the ideas in Example 5, a first implementation could ask another service a_i non-deterministically chosen from a pool $a_1, \ldots, a_n$ to do the job:

$$l :: (\nu a_1 \ldots a_n)\big((\nu av)(\prod_{i=1}^{n} \mathrm{rec}\ X.av!a_i.X \mid *\, a \Rightarrow av?(u).\mathsf{invoke}\ u) \mid \prod_{i=1}^{n} *a_i \Rightarrow P\big)$$

where, instead of directly computing *fun*, upon invocation the service receives (through an intra-site communication on the private channel av) the name of the "private" local service a_i that actually computes *fun*. Notice that these two last transitions are just (non-observable) τ steps and such system is weak bisimilar to system (5). Also, replacing the definitions of a_i with $a_i \Rightarrow P_i$ still yields a system weak bisimilar to the system (5) provided that each P_i is bisimilar to P. On the contrary, removing e.g., the restriction on av breaks the bisimilarity, since the implementation of a could then interact with another channel av in the environment, while the specification does not allow this interaction.

In another possible implementation, a can merge with another session that does the job. For simplicity, we consider just one such session (the case of a nondeterministic choice among many equivalent sessions is analogous):

$$(\nu e)l :: a \Rightarrow \mathrm{rec}\ Y.(\mathsf{merge}^{+}\ e.\mathsf{install}[a \Rightarrow Y]) \mid \mathrm{rec}\ X.(\nu r)r \rhd \mathsf{merge}^{-}\ e.(P|X).$$

In this case, the invocation in the specification is simulated by the invocation in the implementation plus the merge. Notice that e should be bound to avoid other sessions to come into play instead of the wanted one, and that the merge has to be completed before a can be made available again. Similarly, r is restricted to avoid different recursive calls to interfere. Notice that other instances create further nested sessions, but session nesting is immaterial since only the most internal one matters, e.g., $r \rhd r' \rhd P$ is bisimilar to $r' \rhd P$.

5 Related work and concluding remarks

Multiparty sessions are increasingly attracting the attention of researchers in distributed computing. 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 builds on ideas emerged in recent works, but adds to them several original elements. From a technical point of view, μse communication model is inspired by π-calculus [22] and SOC features are based on the SCC [3, 19, 7, 4] family of calculi developed inside the Sensoria project [24]. Multiparty sessions are the main novelty of μse with respect to SCC and they have a strong impact also on other features. For instance, in binary sessions the intended recipient is always understood (the other endpoint of the session), while in multiparty sessions many recipients are possible, and an additional coordination mechanism, such as μse channels, is needed. Also, 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 only the server session is freshly generated. Another difference is that SCC offers more constrained forms of local communication: pipelining [3, 4] and data streaming [19]. In this respect μse is more similar to [7], but its communication primitives exploit the site structure instead of the session structure as the primitives in [7]. We think that this is an important separation of concerns aspect.

In [2] multiparty sessions are considered, but they are required to include one master endpoint and one or more slave endpoints, and direct communication is allowed only between the master and any slave. Our setting is more general since sessions have no predefined structure. The simpler setting of [2] allows a type system based on session types to be defined [15, 16, 8, 10, 13]: developing a similar type system for our generalized setting is more challenging, and is part of our plans for future work. In this respect, also [17, 19, 1, 20, 6] offer a good starting point. Also, [2] uses asynchronous communications, while we use synchronous ones.

The global calculus [8] allows for multiparty 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.

Recently, *global types* have been introduced in [17] in order to describe conversations among several participants; provided that some conditions (e.g., linearity) hold, global types can be projected on and checked against each participant. Several results on disciplined use of global types show how processes reflecting well designed multiparty choreographies enjoy *progress properties* (i.e., well typed processes either terminate or can interact) and *session fidelity* (i.e., well typed processes interactions mimic those specified in their global types). We consider [17] a very inspiring work and we are currently trying to extend the progress and fidelity results to the dynamic setting of $\mu\mathsf{se}$. In fact, dynamic multiparty sessions yield a main difference between $\mu\mathsf{se}$ and the behavioural model adopted in [17] (where the number of participants in a multiparty session is fixed). Actually, safety and liveness properties of dynamic multiparty sessions pose many challenging and interesting research questions. For instance, progress properties should be revisited so that well typed processes should either terminate, interact or eventually allow session merging that do not spoil further interactions.

The intra-session communication model of $\mu\mathsf{se}$ resembles the dyadic synchronisation mechanism of the polyadic pi of [9]. Roughly, the $\mu\mathsf{se}$ process $r \rhd P$ can be seen as the polyadic pi process obtained from P by substituting any occurence of $x(w)$ and $\overline{y}v$ respectively by $r \cdot x(w)$ and $\overline{r \cdot y}\langle v \rangle$. In this respect, the intra-session communication model of $\mu\mathsf{se}$ can be thought as a disciplined use of dyadic synchronisations. An important difference is that $\mu\mathsf{se}$ sessions can be merged via entry points, a feature that would require some form of name fusion on top of [9]. We leave for future work the formal comparison of the two models.

A calculus with coordination mechanism based on event/notification, called XSC, has been introduced in [12] and can model multiparty sessions through a type system. In XSC, components can react to events according to their types that provide a mechanism to associate sessions to events. Though sessions cannot be merged in XSC, its type system permits to correlate events from different sessions. We argue that XSC can be a valid candidate for translating $\mu\mathsf{se}$ in a framework with mechanisms reminiscent of correlation sets.

We conclude by discussing some of the possible extensions of $\mu\mathsf{se}$.

Closing sessions. We plan to extend $\mu\mathsf{se}$ with primitives for explicit session closure, for which nesting of sessions plays a prominent role. In fact, sessions can confine the effect of closure mechanisms so that the part of a running session that must be terminated can be straightforwardly determined.

In the current version of $\mu\mathsf{se}$, session nesting is only exploited as a mechanism for controlling intra-session communication within the same party. For example, P and Q can carry an intra-session interaction neither in $r \rhd (P|Q)$ nor in $r \rhd (P|r \rhd Q)$.

Sophisticated interactions. Communication mechanisms are somehow orthogonal to sessions. In fact, while CCS-like communication [21] is the obvious choice when only two-party sessions are considered, in the presence of multiparty sessions a more natural and more sophisticated alternative would be some variant of multicast (like broadcast [11] or CSP-like interaction [14], or even some combination of different policies [5]). We contend that multiparty sessions as introduced in $\mu\mathsf{se}$ provide a reasonable linguistic background for easily extending the calculus with several sophisticated interaction mechanisms (similarly to what has already been done for graphical languages [18]).

Acknowledgements. Authors thank Nobuko Yoshida for her valuable comments and suggestions, particularly for highlighting some relationships among $\mu\mathsf{se}$ and other proposals.

References

1. L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Festschrift in Honour of Ugo Montanari, on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes in Comput. Sci.* Springer Verlag, 2008. To appear.
2. E. Bonelli and A. Compagnoni. Multisession session types for a distributed calculus. In *Proc. of TGC'07*, volume 4912 of *Lect. Notes in Comput. Sci.*, pages 240–256. Springer Verlag, 2008.
3. 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 *Proc. of WS-FM 2006*, volume 4184 of *Lect. Notes in Comput. Sci.*, pages 38–57. Springer Verlag, 2006.
4. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proc. of FMOODS'08*, Lect. Notes in Comput. Sci. Springer Verlag, 2008. To appear.
5. R. Bruni and I. Lanese. PRISMA: A mobile calculus with parametric synchronization. In *Proc. of TGC'06*, volume 4661 of *Lect. Notes in Comput. Sci.*, pages 132–149. Springer Verlag, 2007.
6. R. Bruni and L. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines, 2008. Submitted.
7. L. Caires, H. T. Vieira, and J. C. Seco. The conversation calculus: A model of service oriented computation. In *Proc. of ESOP'08*, Lect. Notes in Comput. Sci. Springer Verlag, 2008. To appear.
8. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP'07*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 2–17. Springer Verlag, 2007.
9. M. Carbone and S. Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nord. J. Comput.*, 10(2):70–98, 2003.
10. M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A distributed object-oriented language with session types. In *Proc. of TGC'05*, volume 3705 of *Lect. Notes in Comput. Sci.*, pages 299–318. Springer Verlag, 2007.
11. C. Ene and T. Muntean. A broadcast-based calculus for communicating systems. In *Proc. of IPDPS'01*. IEEE Computer Society, 2001.
12. G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto. Coordination via types in an event-based framework. In *Proc. of FORTE'07*, volume 4574 of *Lect. Notes in Comput. Sci.*, pages 66–80, 2007.

13. S. Gay and M. 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 Verlag, 1999.
14. C. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*. Cambridge University Press, 1980.
15. K. Honda. Types for dyadic interaction. In *Proc. of CONCUR'93*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 509–523. Springer Verlag, 1993.
16. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proc. of ESOP'98*, volume 1381 of *Lect. Notes in Comput. Sci.*, pages 22–138. Springer Verlag, 1998.
17. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284, 2008.
18. I. Lanese and E. Tuosto. Synchronized Hyperedge Replacement for heterogeneous systems. In *Proc. of COORDINATION'05*, volume 3454 of *Lect. Notes in Comput. Sci.*, pages 220 – 235. Springer Verlag, 2005.
19. I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM'07*, pages 305–314. IEEE Computer Society Press, 2007.
20. L. Mezzina. How to infer finite session types in a calculus of services and sessions. In *Proc. of COORDINATION'08*, Lect. Notes in Comput. Sci. Springer Verlag, 2008. To appear.
21. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comput. Sci.* Springer Verlag, 1980.
22. 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.
23. OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft.* http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf.
24. Sensoria Project. Software Engineering for Service-Oriented Overlay Computers. Public Web Site. http://sensoria.fast.de/.
25. 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/.