Blind-date Conversation Joining [*]

Luca Cesari[1,2]
[1]Università di Pisa, Italy
cesari@di.unipi.it

Rosario Pugliese[2]
[2]Università degli Studi di Firenze, Italy
luca.cesari@unifi.it
rosario.pugliese@unifi.it

Francesco Tiezzi[3]
[3]IMT Advanced Studies Lucca, Italy
francesco.tiezzi@imtlucca.it

We focus on a form of joining conversations among multiple parties in service-oriented applications where a client may asynchronously join an existing conversation without need to know in advance any information about it. More specifically, we show how the correlation mechanism provided by orchestration languages enables a form of conversation joining that is completely transparent to clients and that we call 'blind-date joining'. We provide an implementation of this strategy by using the standard orchestration language WS-BPEL. We then present its formal semantics by resorting to COWS, a process calculus specifically designed for modelling service-oriented applications. We illustrate our approach by means of a simple, but realistic, case study from the online games domain.

1 Introduction

The increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for supporting automated use. A new computing paradigm has emerged, called Service-Oriented Computing (SOC), that advocates the use of loosely-coupled *services*. These are autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled as the basic blocks for building interoperable and evolvable systems and applications. Currently, the most successful instantiation of the SOC paradigm are *Web Services*, i.e. sets of operations that can be published, located and invoked through the Web via XML messages complying with given standard formats.

In SOC, service definitions are used as templates for creating service instances that deliver application functionalities to either end-user applications or other instances. Upon service invocation, differently from what usually happens in traditional client-server paradigms, the caller (i.e., a service client) and the callee (i.e., a service provider) can engage in a *conversation* during which they exchange the information needed to complete all the activities related to the specific service. For instance, a client of an airplane ticket reservation service usually interacts several times with the service before selecting the specific flight to be reserved. Although initially established between a caller and callee, a conversation can dynamically accommodate and dismiss participants. Therefore, a conversation is typically a loosely-coupled, multiparty interaction among a (possibly dynamically varying) number of participants.

The loosely-coupled nature of SOC implies that, from a technological point of view, the connection between communicating partners cannot be assumed to persist for the duration of a whole conversation. Even the execution of a simple request-response message exchange pattern provides no built-in means of automatically associating the response message with the original request. It is up to each single message to provide a form of context thus enabling partners to associate the message with others. This is achieved by including values in the message which, once located, can be used to correlate the message with others logically forming the same stateful conversation. The link among partners is thus determined by so called *correlation values*: only messages containing the 'right' correlation values are processed by the partner.

Message correlation is an essential part of messaging within SOC as it enables the persistence of activities' context and state across multiple message exchanges while preserving service statelessness and autonomy, and the loosely-coupled nature of SOC systems. It is thus at the basis of Web Services interaction which is implemented on top of stateless Internet protocols, such as the transfer protocol HTTP. For example, the Internet Cookies used by web sites in order to relate an HTTP request to a user profile, thus enabling to return a customized HTML file to the user, are correlation information. Besides being useful to implement stateful communication on top of stateless protocols, correlation is also a flexible and user programmable mechanism for managing loosely-coupled, multiparty conversations. Indeed, correlation data can be communicated to other partners in order to allow them to join a conversation or to delegate the task of carrying out an ongoing conversation.

In this paper, we show another evidence of the flexibility of the correlation mechanism that involves creation of and joining conversations. More specifically, we demonstrate how correlation allows a partner to asynchronously join an existing conversation without need to know in advance any information about the conversation itself, such as e.g. its identifier or the other participants. Since this particular kind of conversation joining is completely transparent to participants we call it *blind-date joining*. It can be used in various domains like e-commerce, events organization, bonus payments, gift lists, etc. In the context of e-commerce, for instance, a social shopping provider (e.g., Groupon [1]) can activate deals only if a certain number of buyers adhere. In this scenario, the blind-date joining can be used to activate these deals. As another example, blind-date joining can be used to organize an event only if a given number of participants is reached. Specifically, the organizers can wait for the right amount of participants and, when this is reached, they send the invitation with the location details to each of them.

We illustrate the blind-date joining strategy by exploiting a simple but realistic case study from the online games domain. We then implement the case study via a well-established orchestration language for web services, i.e. the OASIS standard WS-BPEL [21]. To better clarify and formally present the semantics of the blind-date joining strategy we also introduce a specification of the case study, and its step-by-step temporal evolution, using the process calculus COWS [16, 22], a formalism specifically designed for specifying and combining SOC applications, while modelling their dynamic behaviour.

The rest of the paper is structured as follows. Section 2 surveys syntax and semantics of WS-BPEL and COWS. Section 3 presents how blind-date conversation can be expressed in WS-BPEL by implementing a case study from the online games domain. Section 4 formally describes, by using COWS, the creation and joining phase of blind-date conversations. Finally, Section 5 concludes the paper by also reviewing more closely related work.

2   A glimpse of WS-BPEL and COWS

This section presents a survey of WS-BPEL and COWS. The overview of WS-BPEL gives a high-level description of the aspects captured by its linguistic constructs. Due to lack of space, the overview of COWS gives only a glimpse of its semantics, a full account of which can be found in [22].

2.1   An overview of WS-BPEL

WS-BPEL [21] is essentially a linguistic layer on top of WSDL for describing the structural aspects of Web Services 'orchestration', i.e. the process of combining and coordinating different Web Services to obtain a new, customised service. In practice, and briefly, WSDL [9] is a W3C standard that permits to express the functionalities offered and required by web services by defining, akin object interfaces in

Object-Oriented Programming, the structure of input and output messages of operations.

In WS-BPEL, the logic of interaction between a service and its environment is described in terms of structured patterns of communication actions composed by means of control flow constructs that enable the representation of complex structures. For the specification of orchestration, WS-BPEL provides many different activities that are distinguished between *basic activities* and *structured activities*. Orchestration exploits state information that is stored in variables and is managed through message correlation. In fact, when messages are sent/received, the value of their parameters is stored in variables. Likewise block structured languages, the scope of variables extends to the whole immediately enclosing `<scope>`, or `<process>`, activity (whose meaning is clarified below).

The basic activities are: `<invoke>`, to invoke an operation offered by a Web Service; `<receive>`, to wait for an invocation to arrive; `<reply>`, to send a message in reply to a previously received invocation; `<wait>`, to delay execution for some amount of time; `<assign>`, to update the values of variables with new data; `<throw>`, to signal internal faults; `<exit>`, to immediately end a service instance; `<empty>`, to do nothing; `<compensate>` and `<compensateScope>`, to invoke compensation handlers; `<rethrow>`, to propagate faults; `<validate>`, to validate variables; and `<extensionActivity>`, to add new activity types. Notably, `<reply>` can be combined with `<receive>` to model synchronous request-response interactions.

The structured activities describe the control flow logic of a business process by composing basic and/or structured activities recursively. The structured activities are: `<sequence>`, to execute activities sequentially; `<if>`, to execute activities conditionally; `<while>` and `<repeatUntil>`, to repetitively execute activities; `<flow>`, to execute activities in parallel; `<pick>`, to execute activities selectively; `<forEach>`, to (sequentially or in parallel) execute multiple activities; and `<scope>`, to associate handlers for exceptional events to a primary activity. Activities within a `<flow>` can be further synchronised by means of *flow links*. These are conditional transitions connecting activities to form directed acyclic graphs and are such that a target activity may only start when all its source activities have completed and the condition on the incoming flow links evaluates to true.

The handlers within a `<scope>` can be of four different kinds: `<faultHandler>`, to provide the activities in response to faults occurring during execution of the primary activity; `<compensationHandler>`, to provide the activities to compensate the successfully executed primary activity; `<terminationHandler>`, to control the forced termination of the primary activity; and `<eventHandler>`, to process message or timeout events occurring during execution of the primary activity. If a fault occurs during execution of a primary activity, the control is transferred to the corresponding fault handler and all currently running activities inside the scope are interrupted immediately without involving any fault/compensation handling behaviour. If another fault occurs during a fault/compensation handling, then it is re-thrown, possibly, to the immediately enclosing scope. Compensation handlers attempt to reverse the effects of previously successfully completed primary activities (scopes) and have been introduced to support Long-Running (Business) Transactions (LRTs). Compensation can only be invoked from within fault or compensation handlers starting the compensation either of a specific inner (completed) scope, or of all inner completed scopes in the reverse order of completion. The latter alternative is also called the *default* compensation behaviour. Invoking a compensation handler that is unavailable is equivalent to perform an empty activity.

A WS-BPEL program, also called *(business) process*, is a `<process>`, that is a sort of `<scope>` without compensation and termination handlers.

WS-BPEL uses the basic notion of *partner link* to directly model peer-to-peer relationships between services. Such a relationship is expressed at the WSDL level by specifying the roles played by each of the services in the interaction. This information, however, does not suffices to deliver messages to a

service. Indeed, since multiple instances of the same service can be simultaneously active because service operations can be independently invoked by several clients, messages need to be delivered not only to the correct partner, but also to the correct instance of the service that the partner provides. To achieve this, WS-BPEL relies on the business data exchanged rather than on specific mechanisms, such as WS-Addressing [11] or low-level methods based on SOAP headers. In fact, WS-BPEL exploits *correlation sets*, namely sets of *correlation variables* (called *properties*), to declare the parts of a message that can be used to identify an instance. This way, a message can be delivered to the correct instance on the basis of the values associated to the correlation variables, independently of any routing mechanism.

2.2 An overview of COWS

COWS [16, 22] is a formalism for modelling (and analysing) service-oriented applications. It provides a novel combination of constructs and features borrowed from well-known process calculi such as non-binding receiving activities, asynchronous communication, polyadic synchronization, pattern matching, protection, and delimited receiving and killing activities. As a consequence of its careful design, the calculus makes it easy to model many important aspects of service orchestrations *à la* WS-BPEL, such as service instances with shared state, services playing more than one partner role, stateful conversations made by several correlated service interactions, and long-running transactions. For the sake of simplicity, we present here a fragment of COWS (called μCOWS in [22]) without linguistic constructs for dealing with 'forced termination', since such primitives are not used in this work.

The syntax of COWS is presented in Table 1. We use two countable disjoint sets: the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names* (ranged over by n, m, p, o, ...) mainly used to represent partners and operations. We also use a set of *expressions* (ranged over by ε), whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, values and variables. As a matter of notation, w ranges over values and variables and u ranges over names and variables. Notation $\bar{\cdot}$ stands for tuples, e.g. $\bar{x}$ means $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$) where variables in the same tuple are pairwise distinct. We will omit trailing occurrences of **0**, writing e.g. $p \cdot o? \bar{w}$ instead of $p \cdot o? \bar{w}.\mathbf{0}$, and write $[\langle u_1, \ldots, u_n \rangle] s$ in place of $[u_1] \ldots [u_n] s$. We will write $I \triangleq s$ to assign a name I to the term s.

Services are structured activities built from basic activities, i.e. the empty activity **0**, the invoke activity $_- \cdot _- ! _-$ and the receive activity $_- \cdot _- ? _-$, by means of prefixing $_- . _-$, choice $_- + _-$, parallel composition $_- \mid _-$, delimitation $[_-] _-$ and replication $* _-$. We adopt the following conventions about the operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice.

Invoke and *receive* are the communication activities, which permit invoking an operation offered by a service and waiting for an invocation to arrive, respectively. Besides output and input parameters, both activities indicate an *endpoint*, i.e. a pair composed of a partner name p and an operation name o, through which communication should occur. An endpoint $p \cdot o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. An invoke $p \cdot o! \langle \varepsilon_1, \ldots, \varepsilon_n \rangle$ can proceed as soon as all expression arguments ε_1, ..., ε_n can be successfully evaluated to values. A receive $p \cdot o? \langle w_1, \ldots, w_n \rangle.s$ offers an invocable operation o along a given partner name p and thereafter (due to the *prefixing* operator) the service continues as s. An inter-service communication between these two activities takes place when the tuple of values $\langle v_1, \ldots, v_n \rangle$, resulting from the evaluation of the invoke argument, matches the template $\langle w_1, \ldots, w_n \rangle$ argument of the receive. This causes a substitution of the variables in the receive template (within the scope of variables declarations) with the corresponding values produced by the invoke. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, dynamically re-

Expressions: $\varepsilon, \varepsilon', \ldots$

    Values: $v, v', \ldots$ Variables/Values: $w, w', \ldots$

        Names: $n, m, \ldots$ Variables/Names: $u, u', \ldots$

           Partners: $p, p', \ldots$

            Operations: $o, o', \ldots$

    Variables: $x, y, \ldots$

Services:			Receive-guarded choice:		
s	$::=$		g	$::=$	
	$u \cdot u'!\bar{\varepsilon}$	(invoke)		$\mathbf{0}$	(empty)
	$\mid \quad g$	(receive-guarded choice)		$\mid \quad p \cdot o?\bar{w}.s$	(receive)
	$\mid \quad s \mid s$	(parallel composition)		$\mid \quad g + g$	(choice)
	$\mid \quad [u]\,s$	(delimitation)			
	$\mid \quad *s$	(replication)			

Table 1: COWS syntax

ceived names cannot form the endpoints used to receive further invocations. Indeed, endpoints of receive activities are identified statically because their syntax only allows using names and not variables.

The *empty* activity does nothing, while *choice* permits selecting for execution one between two alternative receives.

Execution of *parallel* services is interleaved. However, if more matching receives are ready to process a given invoke, only one of the receives that generate a substitution with smallest domain (see [22] for further details) is allowed to progress (namely, this receive has an higher priority to proceed than the others). This mechanism permits to model the precedence of a service instance over the corresponding service specification when both can process the same request (see also Section 4).

Delimitation is the only binding construct: $[u]\,s$ binds the element u in the scope s. It is used for two different purposes: to regulate the range of application of substitutions produced by communication, if the delimited element is a variable, and to generate fresh names, if the delimited element is a name.

Finally, the *replication* construct $*s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to implement recursive behaviours and to model business process definitions, which can create multiple instances to serve several requests simultaneously.

3 Blind-date conversations in WS-BPEL

We present here a (web) service capable of arranging matches of 4-players[1] online (card) games, such as e.g. *burraco* or *canasta*. To create or join a match, a player has only to indicate the kind of game he/she would like to play and his/her endpoint reference (i.e., his/her address). Thus, players do not need to know in advance any further information, such as the identifier of the table or the identifiers of other players. Moreover, players do not either know if a new match is created as a consequence of their request to play, or if they join an already existing match. Therefore, the arrangement of tables is *completely transparent* to players.

Figure 1 shows a graphical representation[2] We report below the code of the process where, to sim-

---

[1] It is worth noticing that the blind-date conversation approach works as well with a number of players not fixed a priori. Of course, this would require using some extra activities. Therefore, to better highlight the specificities of the proposed approach, we have preferred to keep the example as simple as possible and, hence, we have fixed the number of players to four.

[2] The representation has been created by using Eclipse BPEL Designer (http://www.eclipse.org/bpel/).
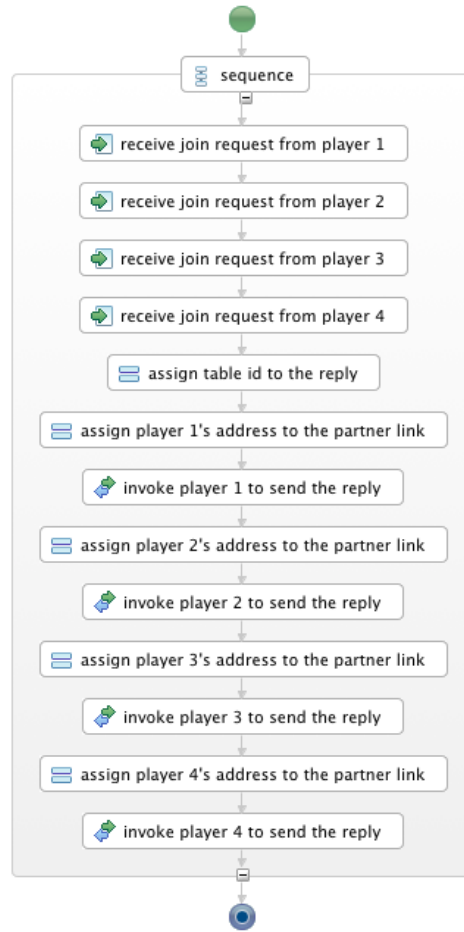
Figure 1: Graphical representation of the WS-BPEL process `TableManager`

plify the reading of the code, we have omitted irrelevant details and highlighted the basic activities `<receive>`, `<invoke>` and `<assign>` by means of a grey background.

```
<process name="TableManager" ... >
<partnerLinks>
  <partnerLink name="tableManager"
               myRole="table" partnerRole="player"
               initializePartnerRole="no"
               partnerLinkType="tableManager:table"/>
</partnerLinks>
<variables>
  <variable messageType="tableManager:request" name="Player1"/>
  <variable messageType="tableManager:request" name="Player2"/>
  <variable messageType="tableManager:request" name="Player3"/>
  <variable messageType="tableManager:request" name="Player4"/>
  <variable messageType="tableManager:reply" name="PlayersReply"/>
</variables>
<correlationSets>
  <correlationSet name="GameCorrelationSet"
                  properties="tableManager:gameNameProperty"/>
</correlationSets>
```

```
<sequence>
  <receive partnerLink="tableManager"
           operation="join"
           variable="Player1"
           createInstance="yes">
      <correlations>
        <correlation initiate="yes" set="GameCorrelationSet"/>
      </correlations>
  </receive>
  <receive partnerLink="tableManager"
           operation="join"
           variable="Player2">
      <correlations>
        <correlation initiate="no" set="GameCorrelationSet"/>
      </correlations>
  </receive>
  <receive partnerLink="tableManager"
           operation="join"
           variable="Player3">
      <correlations>
        <correlation initiate="no" set="GameCorrelationSet"/>
      </correlations>
  </receive>
  <receive partnerLink="tableManager"
           operation="join"
           variable="Player4">
      <correlations>
        <correlation initiate="no" set="GameCorrelationSet"/>
      </correlations>
  </receive>
  <assign>
    ...
    <copy ...>
      <from>getTableID()</from>
      <to>$PlayersReply.payload/tableManager:tableID</to>
    </copy>
  </assign>
  <assign>
    <copy>
      <from>$Player1.payload/tableManager:playerAddress</from>
      <to partnerLink="tableManager"/>
    </copy>
  </assign>
  <invoke inputVariable="PlayersReply"
          operation="start"
          partnerLink="tableManager"/>
  <assign>
    <copy>
      <from>$Player2.payload/tableManager:playerAddress</from>
      <to partnerLink="tableManager"/>
    </copy>
  </assign>
  <invoke inputVariable="PlayersReply"
          operation="start"
          partnerLink="tableManager"/>
```

```
    <assign>
      <copy>
        <from>$Player3.payload/tableManager:playerAddress</from>
        <to partnerLink="tableManager"/>
      </copy>
    </assign>
    <invoke inputVariable="PlayersReply"
            operation="start"
            partnerLink="tableManager"/>
    <assign>
      <copy>
        <from>$Player4.payload/tableManager:playerAddress</from>
        <to partnerLink="tableManager"/>
      </copy>
    </assign>
    <invoke inputVariable="PlayersReply"
            operation="start"
            partnerLink="tableManager"/>
  </sequence>
</process>
```

The process uses only one partner link, namely `tableManager`, that provides two roles: `table` and `player`. The former is used by the process to receive the players' requests, while the latter is used by players to receive the table identifier. Five variables are used for storing data of the exchanged messages: one for each player request and one for the manager response. Notably, the used message style[3] is *document*, thus messages are formed by a single part, called `payload`, that contains all message data. Therefore, we use XPath expressions of the form `$VariableName.payload/Path` to extract or store data in message variables.

The process starts with a `<receive>` activity waiting for a message from a player; the message contains a request (stored in the variable `Player1`) to participate to a match of a given game. Whenever prompted by a player's request, the process creates an instance (see the option `createInstance="yes"` in the first `<receive>`), corresponding to a new (virtual) card-table of the game specified by the player, and is immediately ready to concurrently serve other requests. Service instances are indeed the WS-BPEL counterpart of inter-service conversations. In order to deliver each request to an existing instance corresponding to a table of the requested game (if there exists one), the name of the game is used as correlation datum. Thus, each `<receive>` activity specifies the correlation set `GameCorrelationSet`, which is instantiated by the initial `<receive>`, in order to receive only requests for the same game indicated by the first request. The `<property>` defining the correlation set is declared in the WSDL document associated to the process as follows:

```
<prop:property name="gameNameProperty" type="xs:string"/>

<prop:propertyAlias messageType="this:request" part="payload"
                    propertyName="this:gameNameProperty">
  <prop:query queryLanguage=...>//this:RequestElement/this:gameName</prop:query>
</prop:propertyAlias>
```

---

[3] The SOAP message style configuration is specified in the *binding* section of the WSDL document associated to the WS-BPEL process. We have preferred to use the document-style rather than the RPC-style, because the former minimizes coupling between the interacting parties.

A `<property>` specifies an element of a correlation set and relies on one (or more) `<property-Alias>` to identify correlation values inside messages. In our specification, the `<propertyAlias>` extracts from `<request>` messages the needed element by using an XPath `<query>`. Then, the correlation set `GameCorrelationSet` is defined by the property `gameNameProperty` that identifies the string element `gameName` of the messages sent by players.

Once the initial `<receive>` is executed and an instance is created, other three `<receive>` activities are sequentially performed by such instance, in order to complete the card-table for the new match. Notice that the correlation mechanism ensures that only players that want to play the same game are put together in a table.

When four players join a conversation for a new match (which, in WS-BPEL, corresponds to a process instance), a unique table identifier is generated, by means of the custom XPath expression `getTableID()`, and inserted into the variable `PlayersReply`. This variable contains the message that will be sent back to each player via four `<invoke>` activities using the `tableManager` partner link. Before every `<invoke>`, an activity `<assign>` is executed to extract (by means of an XPath expression) the endpoint reference of the player, contained within the player's request, and to store it into the `partnerRole` of the `tableManager` partner link. These assignments allow the process to properly reply in an asynchronous way to the players.

Now, the new table is arranged and, therefore, the players can start to play by using the received table identifier and by interacting with another service dedicated to this purpose (which, of course, is out of the scope of this work).

Experimenting with the WS-BPEL process.   This WS-BPEL process can be experimented via the Web interface at `http://reggae.dsi.unifi.it/blinddatejoining/`, or by downloading from the same address the source and binary code.

More specifically, the process is configured to be deployed in a WS-BPEL engine Apache ODE [3]. Indeed, we have equipped the process with the corresponding WSDL and deployment descriptor files, which provide typing and binding information, respectively. Notably, in order to call the custom XPath expression `getTableID()` within the process, its definition must be previously installed in the ODE engine as a Java library. For the sake of simplicity, we have defined such expression as a random function.

To experiment with the WS-BPEL program, we have developed a sort of testing environment consisting of a few Java classes implementing the service clients. Such classes rely on the artifacts automatically generated by JAX-WS [10] from the WSDL document of the process, and simply exchange SOAP messages with the process. These clients are instantiated and executed by a Web application developed by using the Play framework [25]; a screenshot of such application is shown in Figure 2. Notably, players created in a given browser session could be assigned to tables together with players created in other sessions.

4   Semantics mechanisms underlying blind-date conversation joining

In this section, to clarify the behaviour at runtime of the `TableManager` process (and of its instances), we formally specify a scenario involving the manager service by means of the process calculus COWS. The aim is to shed light on the effect of the blind-date joining and to show how it can be easily programmed through the correlation approach. Moreover, this formal account also permits clarifying the mechanisms underlying message correlation (i.e., shared input variables, pattern-matching and priority among receive activities).
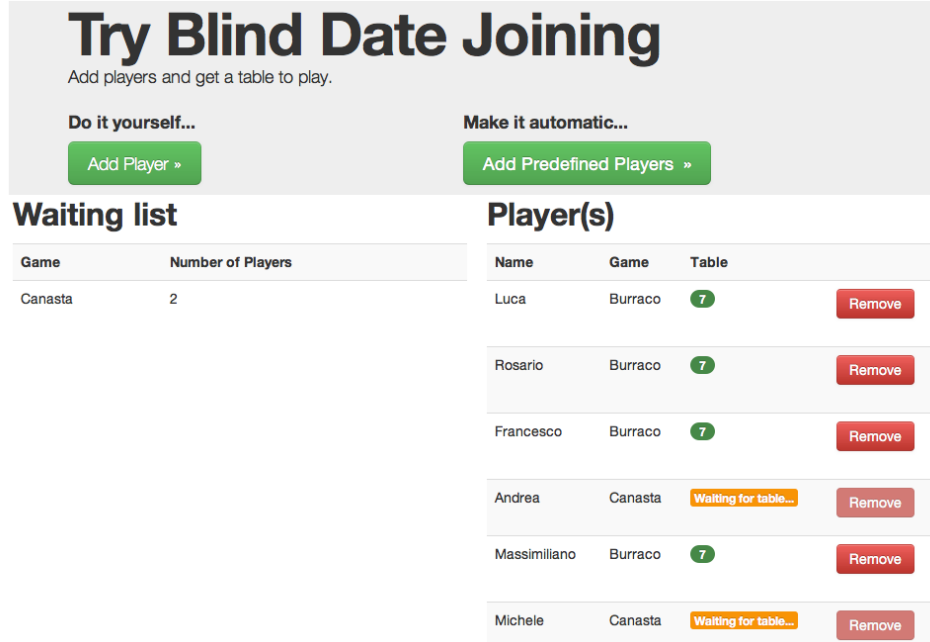
Figure 2: A screenshot of the Web application for experimenting with the process `TableManager`

The `TableManager` process can be rendered in COWS as follows:

$$
\begin{aligned}
TableManagerProcess \triangleq\ &*\left[x_{game}, x_{player1}, x_{player2}, x_{player3}, x_{player4}\right] \\
&manager \bullet join?\langle x_{game}, x_{player1}\rangle. \\
&manager \bullet join?\langle x_{game}, x_{player2}\rangle. \\
&manager \bullet join?\langle x_{game}, x_{player3}\rangle. \\
&manager \bullet join?\langle x_{game}, x_{player4}\rangle. \\
&\left[tableId\right]\left(x_{player1} \bullet start!\langle tableId\rangle\right. \\
&\qquad\qquad\quad \mid x_{player2} \bullet start!\langle tableId\rangle \\
&\qquad\qquad\quad \mid x_{player3} \bullet start!\langle tableId\rangle \\
&\qquad\qquad\quad \left.\mid x_{player4} \bullet start!\langle tableId\rangle\right)
\end{aligned}
$$

The replication operator $*$ specifies that the above term represents a service definition, which acts as a persistent service capable of creating multiple instances to simultaneously serve concurrent requests. The delimitation operator $[\,]$ declares the scope of variables x_{game} and $x_{playeri}$, with $i = 1..4$. The endpoint $manager \bullet join$ (composed by the partner name $manager$ and the operation $join$) is used by the service to receive join requests from four players. When sending their request, the players are required to provide only the kind of game, stored in x_{game}, and their partner names, stored in $x_{playeri}$, that they will then use to receive the table identifier. Player's requests are received through receive activities of the form $manager \bullet join?\langle x_{game}, x_{playeri}\rangle$, which are correlated by means of the shared variable x_{game}. Then, the delimitation operator is used to create a fresh name $tableId$ that represents an unique table identifier. Such identifier will be then communicated to each player by means of four invoke activities $x_{playeri} \bullet start!\langle tableId\rangle$. Notably, differently from the WS-BPEL specification of the process, in the COWS definition the assign activities are not necessary, because their role is played by the substitutions generated by the interactions along the endpoint $manager \bullet join$.

Consider now the following system

$$Luca \mid Rosario \mid Francesco \mid \ldots \mid TableManagerProcess$$

where the players are defined as follows

$$Luca \triangleq manager \cdot join!\langle burraco, p_L \rangle \mid [x_{id}]\, p_L \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Luca \rangle$$

$$Rosario \triangleq manager \cdot join!\langle canasta, p_R \rangle \mid [x_{id}]\, p_R \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Rosario \rangle$$

$$Francesco \triangleq manager \cdot join!\langle burraco, p_F \rangle \mid [x_{id}]\, p_F \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Francesco \rangle$$

$$\ldots$$

If *Luca* requests to join a match, since there are not tables under arrangement, *TableManager-Process* initialises a new match instance (highlighted by grey background) and the system evolves to:

$$[x_{id}]\, p_L \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Luca \rangle$$
$$\mid Rosario \mid Francesco \mid \ldots \mid TableManagerProcess$$
$$\mid\ [x_{player2}, x_{player3}, x_{player4}]$$
$$\quad manager \cdot join?\langle burraco, x_{player2} \rangle.$$
$$\quad manager \cdot join?\langle burraco, x_{player3} \rangle.$$
$$\quad manager \cdot join?\langle burraco, x_{player4} \rangle.$$
$$\qquad [tableId]\, (\, p_L \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player2} \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player3} \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player4} \cdot start!\langle tableId \rangle\,)$$

Now, if *Rosario* invokes *TableManagerProcess*, a second match instance (highlighted by dark grey background) is created:

$$[x_{id}]\, p_L \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Luca \rangle$$
$$\mid [x_{id}]\, p_R \cdot start?\langle x_{id} \rangle. \langle \text{rest of } Rosario \rangle$$
$$\mid Francesco \mid \ldots \mid TableManagerProcess$$
$$\mid\ [x_{player2}, x_{player3}, x_{player4}]$$
$$\quad manager \cdot join?\langle burraco, x_{player2} \rangle.$$
$$\quad manager \cdot join?\langle burraco, x_{player3} \rangle.$$
$$\quad manager \cdot join?\langle burraco, x_{player4} \rangle.$$
$$\qquad [tableId]\, (\, p_L \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player2} \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player3} \cdot start!\langle tableId \rangle$$
$$\qquad\qquad\qquad \mid x_{player4} \cdot start!\langle tableId \rangle\,)$$

$$| \quad [x_{player2}, x_{player3}, x_{player4}]$$
$$manager \cdot join?\langle canasta, x_{player2} \rangle.$$
$$manager \cdot join?\langle canasta, x_{player3} \rangle.$$
$$manager \cdot join?\langle canasta, x_{player4} \rangle.$$
$$[tableId'] \, (\, p_R \cdot start!\langle tableId' \rangle$$
$$| \, x_{player2} \cdot start!\langle tableId' \rangle$$
$$| \, x_{player3} \cdot start!\langle tableId' \rangle$$
$$| \, x_{player4} \cdot start!\langle tableId' \rangle \,)$$

When *Francesco* invokes *TableManagerProcess*, the process definition and the first created instance, being both able to receive the same message $\langle burraco, p_F \rangle$ along the endpoint *manager · join*, compete for the request $manager \cdot join!\langle burraco, p_F \rangle$. COWS's (prioritized) semantics precisely establishes how this sort of race condition is dealt with: only the existing instance is allowed to evolve, as required by WS-BPEL. This is done through the dynamic prioritised mechanism of COWS, i.e. assigning the receives performed by instances (having a more defined pattern and requiring less substitutions) a greater priority than the receives performed by a process definition. In fact, in the above COWS term, the first instance can perform a receive matching the message and containing only one variable in its argument, while the initial receive of *TableManagerProcess* contains two variables. In this way, the creation of a new instance is prevented. Moreover, pattern-matching permits delivering the request to the appropriate instance, i.e. that corresponding to a *burraco* match. Therefore, the only feasible computation leads to the following term

$$[x_{id}] \, p_L \cdot start?\langle x_{id} \rangle. \, \langle \text{rest of } Luca \rangle$$
$$| \, [x_{id}] \, p_R \cdot start?\langle x_{id} \rangle. \, \langle \text{rest of } Rosario \rangle$$
$$| \, [x_{id}] \, p_F \cdot start?\langle x_{id} \rangle. \, \langle \text{rest of } Francesco \rangle$$
$$| \, \dots \, | \, TableManagerProcess$$
$$| \quad [x_{player3}, x_{player4}]$$
$$manager \cdot join?\langle burraco, x_{player3} \rangle.$$
$$manager \cdot join?\langle burraco, x_{player4} \rangle.$$
$$[tableId] \, (\, p_L \cdot start!\langle tableId \rangle$$
$$| \, p_F \cdot start!\langle tableId \rangle$$
$$| \, x_{player3} \cdot start!\langle tableId \rangle$$
$$| \, x_{player4} \cdot start!\langle tableId \rangle \,)$$
$$| \quad [x_{player2}, x_{player3}, x_{player4}]$$
$$manager \cdot join?\langle canasta, x_{player2} \rangle.$$
$$manager \cdot join?\langle canasta, x_{player3} \rangle.$$
$$manager \cdot join?\langle canasta, x_{player4} \rangle.$$
$$[tableId'] \, (\, p_R \cdot start!\langle tableId' \rangle$$
$$| \, x_{player2} \cdot start!\langle tableId' \rangle$$
$$| \, x_{player3} \cdot start!\langle tableId' \rangle$$
$$| \, x_{player4} \cdot start!\langle tableId' \rangle \,)$$

where *Francesco* joined to the *burraco* table under arrangement.

Eventually, with the arrival of other requests from players that want to play *burraco*, the *TableManagerProcess* completes to arrange the *burraco* table and contacts the players for communicating them the table identifier:

$$[\textit{tableId}] \, (\, \langle \text{rest of } \textit{Luca} \rangle \mid \langle \text{rest of } \textit{Francesco} \rangle \mid \dots)$$
$$\mid [x_{id}] \, p_R \bullet \textit{start}?\langle x_{id} \rangle. \langle \text{rest of } \textit{Rosario} \rangle$$
$$\mid \dots \mid \textit{TableManagerProcess}$$
$$\mid [x_{player2}, x_{player3}, x_{player4}]$$
$$\quad \textit{manager} \bullet \textit{join}?\langle \textit{canasta}, x_{player2} \rangle.$$
$$\quad \textit{manager} \bullet \textit{join}?\langle \textit{canasta}, x_{player3} \rangle.$$
$$\quad \textit{manager} \bullet \textit{join}?\langle \textit{canasta}, x_{player4} \rangle.$$
$$\quad\quad [\textit{tableId}'] \, (\, p_R \bullet \textit{start}!\langle \textit{tableId}' \rangle$$
$$\quad\quad\quad\quad \mid x_{player2} \bullet \textit{start}!\langle \textit{tableId}' \rangle$$
$$\quad\quad\quad\quad \mid x_{player3} \bullet \textit{start}!\langle \textit{tableId}' \rangle$$
$$\quad\quad\quad\quad \mid x_{player4} \bullet \textit{start}!\langle \textit{tableId}' \rangle \,)$$

Therefore, the players of table *tableId* (including *Luca* and *Francesco*) can start to play, while *Rosario* continues to wait for other *canasta* players.

5   Concluding remarks

We have illustrated blind-date conversation joining, a strategy allowing a participant to join a conversation without need to know information about the conversation itself, such as e.g. its identifier or the other participants, but the endpoint of the service provider. This is possible because some values in the request define which conversation the participant will join. In case such a conversation does not exist, a new one associated to these values will be created.

We showed how the blind-date conversation joining strategy can be implemented in the well-established industrial standard WS-BPEL. In order to accomplish this task, we used WS-BPEL correlation sets to filter the client requests and create conversations identified by means of the filtered values.

We also presented a formal semantics for the strategy by using the process calculus COWS. The semantics permits to highlight what the effect of the strategy is, by clarifying the mechanism underlying message correlation and showing how the strategy can be easily programmed in a correlation-based environment. This also permits appreciating the conciseness and cleanness of the COWS specification of the `TableManager` process, especially when compared with its WS-BPEL counterpart.

We developed a case study to show how our strategy can be used in a realistic scenario. The case study defines an online games provider that arranges online matches of 4-players card games. This scenario is implemented using WS-BPEL and executed on Apache ODE. As we showed in Section 2.1, despite the XML markup used by WS-BPEL, the blind-date joining strategy permits to have a smooth and clean implementation. We also equipped the case study with a 'testing environment' in order to facilitate its testing.

Related work.   The peculiar form of conversation joining studied in this paper, which we call 'blind-date', originates from the message correlation mechanism used for delivering messages to the appropriate

service instances in both orchestration languages and formalisms for SOC. This joining strategy is, at least in principle, independent from the specific language or formalism used to enact it. In this paper, we have used the language WS-BPEL and the formalism COWS, but different choices could have been made. For example, Jolie [20] could be used as correlation-based orchestration language and SOCK [12] as the formalism, being the former a Java-based implementation of the latter. However, our choice fell on WS-BPEL because it is an OASIS open standard well-accepted by industries and, hence, supported by well-established and maintained engines. Instead, COWS has been selected because of its strict correspondence with WS-BPEL while, at the same time, being a core calculus consisting of just a few constructs, which makes it more suitable than WS-BPEL to reason on applications' behaviour.

In the SOC literature, two main approaches have been considered to connect the interaction protocols of clients and of the respective service instances. That based on the correlation mechanism was first exploited in [24] where, however, only interaction among different instances of a single service are taken into account. Another correlation-based formalism, besides COWS and SOCK mentioned above, is the calculus *Corr* [18], which is a sort of value-passing CCS without restriction and enriched with constructs for expressing services and their instances. *Corr* has been specifically designed to capture behaviours related to correlation aspects in a specific WS-BPEL engine (namely, ActiveBPEL [2]). Another work with an aim similar to ours, i.e. to show an exploitation of the correlation-based mechanism for dealing with issues raised by practical scenarios, is presented in [17]. Such work proposes an implementation of a correlation-based primitive allowing messages to be delivered to more than one service conversation. Anyway, WS-BPEL (and COWS as well) natively provides such kind of broadcast primitive.

A large strand of work, instead, relies on the explicit modelling of interaction *sessions* and their dynamic creation. A session corresponds to a private channel (*à la* π-calculus [19]) which is implicitly instantiated when calling a service: it binds caller and callee and is used for their future conversation. Although this the technology underling SOC, its abstraction level has proved convenient for reasoning about SOC applications. Indeed, session-based conversation can be regulated by so called *session types* that can statically guarantee a number of desirable properties, such as communication safety, progress, predictability. scenario declared in the session type). Therefore, an important group of calculi for modeling and proving properties of services is based on the explicit notion of interaction session. Most of such work (among which we mention [13, 7, 15, 4]) has been devoted to study *dyadic* sessions, i.e. interaction sessions between only two participants.

Recently, another body of work (as, e.g., [14, 5, 6, 8]) focussed on a more general form of sessions, called *multiparty* sessions/conversations, which are closer to the notion of conversation that we have considered in this paper. The multiparty session approach proposed in [14] permits expressing a conversation by means of channels shared among the participants. However, the conversation is created through a single synchronization among all participants, whose number is fixed at design time. This differs from our notion of blind-date joining, where once a conversation has been created the participants can asynchronously join. Moreover, although in our case study we consider a conversation of five participants, in principle this number can change at runtime. The μse language [5] permits the declaration of multiparty sessions in a way transparent to the user. However, rather than relying on the correlation mechanism, as in our approach, μse creates multiparty conversations by using a specific primitive that permits to merge together previously created conversations. Instead, when relying on correlation, the conversation merging is automatically performed for each correlated request. Another formalism dealing with multiparty interactions is the Conversation Calculus [23, 6]. It uses the conversation-based mechanism for inter-session communication, which permits to progressively accommodate and dismiss participants from the same conversation. This is realized by means of named containers for processes, called conversation contexts. However, processes in unrelated conversations cannot interact directly. Moreover, conversa-

tion joining is not transparent to a new participant, because he has to know the name of the conversation. Instead, in our correlation-based approach, a conversation is represented by a service instance, which is not accessed via an identifier, but via the correlation values specified by the correlation set.

In conclusion, the lower-level mechanism based on correlation sets, that exploits business data to correlate different interactions, is more robust and fits the loosely-coupled world of Web Services better than that based on explicit session references. It turned out to be powerful and flexible enough to fulfill the need of SOC, e.g. it easily allows a single message to participate in multiple conversations, each (possibly) identified by separate correlation values, which instead requires non-trivial workarounds by using the session-based approach. It is not a case that also the standard WS-BPEL uses correlation sets.

Future work.   To better highlight the semantics mechanisms underlying blind-date conversation joining, we have preferred to consider a quite simple case study. Anyway, many other features could be added without much effort that would make the case study even more realistic. For example, a player should be able to play concurrently in more than one match. However, for the sake of simplicity, we assume here that a player can ask to play the same game more than once only if each time it waits for the response to the previous request, otherwise it could be assigned more than once to the same table. Moreover, *TableManagerProcess* can be easily tailored to arrange matches for n-players online games with $n \neq 4$ (e.g. poker, bridge, . . .). It could also be modified so to be able to manage games with a number of players not statically known, as well as games that can be played by a variable number of players, i.e. games that may begin with a minimum number of players but later on other players can dynamically join and some others can stop playing and leave. The fact that some players may give up, i.e. the number of players may also decrease, implies that some tables may dynamically have not enough players for the game to go on. In this case, tables with an insufficient number of players could be joined and there are several possible strategies to to this: a centralized solution could exploit the table manager as a forwarder for the player messages, while a decentralized one could exploit the tables with not enough players as forwarders; alternatively, we could exploit *memory cell* services to store the (dynamically updatable) value of the table assigned to each single player thus avoiding to replace, one and for all, the correlation variable with this value; finally, *TableManagerProcess* instances, instead of single tables, could represent rooms containing more tables for the same game and manage all the requests for that game. In order to avoid deadlocks, we could add timeouts to receive activities of *TableManagerProcess* and players' services. This can be done both in WS-BPEL, by exploiting `onAlarm` events within `pick` activities, and in COWS, by using *wait* activities in conjunction with choice activities (as shown in [22]).

We leave as a future work the task of formalizing the pattern underlying blind-date conversation joining in a way which is independently from the specific case study considered in this paper. We also plan to study and develop type theories for describing and regulating the correlation mechanism. This will permit to guarantee desired properties like those defined for dyadic and multiparty session types. We expect that to develop type theories for ensuring properties of the correlation mechanism be more difficult than doing so for session-based interaction because the communication media are not necessarily private and the correlation values are determined at runtime.

References

[1] *Groupon.* Web site: `http://www.groupon.com/`.

[2] Active Endpoints (2008): *ActiveBPEL 5.0.2.* Available at `http://www.activevos.com/community-open-source.php`.

[3]   Apache Software Foundation (2011): *Apache ODE 1.3.5*. Available at `http://ode.apache.org/`.

[4]   M. Boreale, R. Bruni, R. De Nicola & M. Loreti (2008): *Sessions and Pipelines for Structured Service Programming*. In: *FMOODS*, *LNCS* 5051, Springer, pp. 19–38.

[5]   R. Bruni, I. Lanese, H.C. Melgratti & E. Tuosto (2008): *Multiparty Sessions in SOC*. In: *COORDINATION*, *LNCS* 5052, Springer, pp. 67–82.

[6]   L. Caires & H.T. Vieira (2010): *Conversation types*. *Theor. Comput. Sci.* 411(51-52), pp. 4399–4440.

[7]   M. Carbone, K. Honda & N. Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *ESOP*, *LNCS* 4421, Springer, pp. 2–17.

[8]   M. Carbone & F. Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In: *POPL*, ACM, pp. 263–274.

[9]   E. Christensen, F. Curbera, G. Meredith & S. Weerawarana (2001): *Web Services Description Language (WSDL) 1.1*. Technical Report, W3C. Available at `http://www.w3.org/TR/wsdl/`.

[10] GlassFish community (2012): *JAX-WS 2.2.7*. Available at `http://jax-ws.java.net/`.

[11] M. Gudgin, M. Hadley & T. Rogers (2006): *Web Services Addressing 1.0 - Core*. Technical Report, W3C. W3C Recommendation.

[12] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi & G. Zavattaro (2006): *SOCK: A Calculus for Service Oriented Computing*. In: *ICSOC*, *LNCS* 4294, Springer, pp. 327–338.

[13] K. Honda, V. T. Vasconcelos & M. Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, *LNCS* 1381, Springer, pp. 122–138.

[14] K. Honda, N. Yoshida & M. Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, ACM Press, pp. 273–284.

[15] I. Lanese, F. Martins, A. Ravara & V.T. Vasconcelos (2007): *Disciplining Orchestration and Conversation in Service-Oriented Computing*. In: *SEFM*, IEEE Computer Society Press, pp. 305–314.

[16] A. Lapadula, R. Pugliese & F. Tiezzi (2007): *A Calculus for Orchestration of Web Services*. In: *ESOP*, *LNCS* 4421, Springer, pp. 33–47.

[17] J. Mauro, M. Gabbrielli, C. Guidi & F. Montesi (2011): *An Efficient Management of Correlation Sets with Broadcast*. In: *COORDINATION*, *LNCS* 6721, Springer, pp. 80–94.

[18] H.C. Melgratti & C. Roldán (2012): *On Correlation Sets and Correlation Exceptions in ActiveBPEL*. In: *TGC*, *LNCS* 7173, Springer, pp. 212–226.

[19] R. Milner, J. Parrow & D. Walker (1992): *A Calculus of Mobile Processes, I and II*. Information and Computation 100(1), pp. 1–40, 41–77.

[20] F. Montesi, C. Guidi, R. Lucchi & G. Zavattaro (2007): *JOLIE: a Java Orchestration Language Interpreter Engine*. In: *MTCoord*, *Electronic Notes in Theoretical Computer Science* 181, Elsevier, pp. 19–33.

[21] OASIS WSBPEL TC (2007): *Web Services Business Process Execution Language Version 2.0*. Technical Report, OASIS. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[22] R. Pugliese & F. Tiezzi (2012): *A calculus for orchestration of web services*. *J. Applied Logic* 10(1), pp. 2–31.

[23] H.T. Vieira, L. Caires & J. C. Seco (2008): *The Conversation Calculus: A Model of Service-Oriented Computation*. In: *ESOP*, *LNCS* 4960, Springer, pp. 269–283.

[24] M. Viroli (2004): *Towards a Formal Foundation to Orchestration Languages*. In: *WS-FM*, *Electronic Notes in Theoretical Computer Science* 105, Elsevier, pp. 51–71.

[25] Zenexity and Typesafe (2013): *Play framework 2.1.0*. Available at `http://www.playframework.com/`.