SOCA
DOI 10.1007/s11761-017-0214-0

ORIGINAL RESEARCH PAPER

Blind-date conversation joining

Luca Cesari[1] · **Rosario Pugliese**[2] · **Francesco Tiezzi**[3]

Received: 7 December 2016 / Revised: 6 June 2017 / Accepted: 14 June 2017

Abstract In service-oriented applications, service providers and their clients can engage in conversations to exchange the data required to achieve their business goals. In this paper, we focus on a particular kind of conversation joining, which we call *blind-date*, where a client may join a conversation among multiple parties in an asynchronous and completely transparent way. Indeed, the client can join the conversation without knowing any information about it in advance. More specifically, we show that the correlation mechanism provided by orchestration languages enables the blind-date conversation joining strategy. To demonstrate the feasibility of the approach, we provide an implementation of this strategy by using the standard orchestration language WS- BPEL. Moreover, to clarify the run-time effects of the blind-date joining, we formally describe its behaviour by resorting to COWS, a process calculus specifically designed for modelling service-oriented applications. We illustrate our approach by means of a simple example and a more realistic case study from the online games domain.

Keywords Service-oriented computing · Conversations · Message correlation · Formal modelling

✉ Francesco Tiezzi
francesco.tiezzi@unicam.it

Luca Cesari
luca@cesari.me

Rosario Pugliese
rosario.pugliese@unifi.it

[1] RCP Vision, Florence, Italy

[2] Università degli Studi di Firenze, Florence, Italy

[3] Università di Camerino, Camerino, Italy

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 is *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, while the callee can concurrently serve other requests. 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 the caller and the callee, a conversation can dynamically accommodate and dismiss participants. Therefore, a conversation is typically a series of loosely coupled, multiparty interactions 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 com-

Published online: 12 July 2017

municating 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 for 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 conversation. The link among partners is thus determined by the so-called *correlation values*: only messages containing the 'right' correlation values are processed by a partner.

Message correlation is an essential part of messaging within SOC as it enables context and state persistence of activities across multiple message exchanges, while preserving service statelessness and autonomy, and the loosely coupled nature of SOC systems. It is hence at the basis of Web services interaction, which is implemented on top of stateless Internet protocols, such as the transfer protocol HTTP. An example of correlation information are the Internet cookies. They are used by websites in order to relate an HTTP request to a user profile, so to be able to return a customised HTML file to the user. 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 them to carry out an ongoing conversation.

In this paper, we show another evidence of the flexibility of the message correlation mechanism that involves creation of conversations and subsequent joining of partners. More specifically, we demonstrate how correlation allows a partner to asynchronously join an existing conversation without the need to know in advance information about the conversation itself, such as the conversation identifier or the identity of 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 such as e-commerce, events organisation, bonus payments, gift lists. In the context of e-commerce, for instance, a social shopping provider (e.g. Groupon [17]) 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 organise an event only if a given number of participants is reached. Specifically, the organisers 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.

The main contribution of this work is the characterisation of the blind-date joining strategy, a concept not studied yet in the SOC literature, and its practical realisation in terms of typical mechanisms provided by orchestration languages. In particular, the specific contributions of the paper are as follows:

- presentation of the general concepts underlying the blind-date joining strategy (Sect. 2);
- implementation of the blind-date joining strategy, illustrated on an instantiation from the online games domain, via a well-established orchestration language for web services, i.e. the OASIS standard WS- BPEL [35] (Sect. 4);
- formalisation of the semantics of the blind-date joining strategy by means of a specification of the case study, and its step-by-step temporal evolution, using the process calculus COWS [23,37], a formalism specifically designed for specifying and combining SOC applications, while modelling their dynamic behaviour (Sect. 5).
- illustration of the effectiveness of the blind-date conversation approach through a more realistic conversation-based service (Sect. 6).

For the reader's convenience, in Sect. 3 we survey syntax and semantics of WS- BPEL and COWS. In Sect. 7 we review more closely related work. Finally, in Sect. 8 we conclude by also touching upon a few directions for future work.

This work is an extended and revisited version of our former development introduced in [10]. Here we have added a general presentation of the blind-date conversation joining strategy and enucleated its most relevant features. This presentation is independent from the online game case study. We have also added an extended implementation of the case study, going towards a more realistic application. Finally, we have revised and extended the discussion of related work.

2 Blind-date conversations

Parties in SOC, i.e. service providers and clients, engage in *conversations* consisting of interactions among each other in order to exchange all information required to obtain/provide services and to achieve the related goals. A conversation is initially established by two parties, but then other participants can join it. Typically, a conversation actually starts when a given number of participants is reached.

In this paper, we are mainly interested in the creation and joining phase of conversations, rather than in their progress and completion. In particular, we focus on a specific kind of conversation that we call *blind-date*. Parties willing to participate in such conversations do not need to know in advance any low-level information concerning the conversation itself (e.g. a unique conversation identifier) or concerning the other participants (e.g. their identities or communication endpoints). In other words, creation of and joining these conversations is completely transparent to their participants. Actually, parties are also not required to know if a new conversation is created

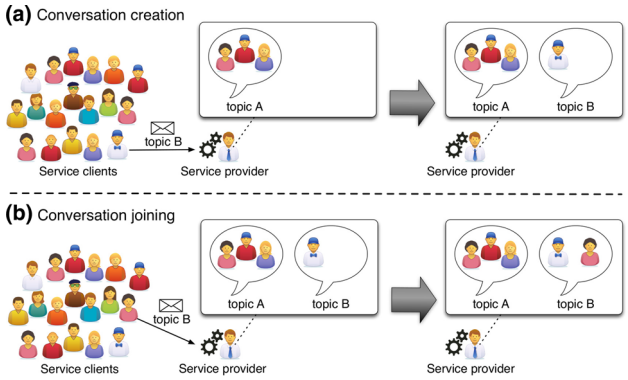

Fig. 1 Blind-date conversation: creation and joining

as a consequence of their request to participate, or if they join an already existing conversation.

On the other hand, to create/join an appropriate conversation, parties have to specify the information concerning 'what the conversation is talking about', that we call *conversation topic*. In fact, in a blind-date setting, a party is not interested in participating in a specific conversation of a given service, but just to one of its conversations that concern the desired topic. For example, in the case study considered in this paper, client parties are players that want to play online (card) games, while the provider party is a service that arranges their matches. In this case, the topic of the conversation is simply the name of the game, because player parties, according to the blind-date strategy, do not mind who are the other players of a match. It is worth noticing that conversation topics are *public* data, already known by all potential participants that, thus, do not need to know and interact each other to agree on the values of topics.

Figure 1 graphically shows a typical scenario where a conversation is created and joined. When a participation request concerning topic B is received by the service provider, it checks if there already exists a conversation about such topic that is available to host other participants. If such a conversation does not exist (Figure 1.a), a new conversation, containing only the initiator participants and implicitly the service itself, is created. Instead, if such a conversation already exists (Figure 1.b), the creation of a new conversation is prevented and the party joins an available one. Therefore, the blind-date conversation joining strategy requires mechanisms for associating participation requests with conversations (according to their topics) and for preventing the creation of a new conversation when an appropriate conversation is already available.

To sum up, the main ingredients of the blind-date conversation joining strategy are as follows:

1. publicly known conversation topics;
2. topic-based association of participation requests to conversations;
3. precedence of available conversations over creation of new conversations.

Our aim is to show how this strategy can be conveniently realised by relying on the message correlation mechanism provided by many SOC programming languages, in particular service orchestration languages (see, for example, [19,23,25,34,35,37]). In fact, although the management of blind-date conversations could be implemented by means of low-level ad hoc code, we believe that the use of high-level standard mechanisms specifically devised for SOC programming makes the implementation task easier and the proposed solution more loosely coupled and portable.

As mentioned in Introduction and explained in more detail in the rest of the paper, message correlation simply consists in including correlation values in messages and using such values, in a pattern-matching fashion, to associate messages with others belonging to the same conversation. Thus, correlation values directly correspond to the conversation topics (ingredient 1 required by the blind-date conversation joining strategy) and the pattern-matching mechanism to the topic-based association (ingredient 2). Moreover, in SOC languages capable of dealing with multiple start activities[1], the correlation mechanism is implemented in such a way that service instances have higher priority than the corresponding service definitions to receive correlated messages. This feature of correlation permits ensuring that available conversations take precedence over the creation of new ones (ingredient 3).

Therefore, the correlation mechanism already present in many SOC languages provides all the ingredients required by the blind-date conversation joining strategy.

3 A glimpse of WS-BPEL and COWS

This section overviews WS- BPEL and COWS by providing an intuitive description of the aspects captured by their linguistic constructs. A detailed account of WS- BPEL can be found in [35], while a formal presentation of COWS semantics is in [37].

3.1 An overview of WS-BPEL

WS- BPEL [35] 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 coordi-

[1] In case of multiple start activities [35, Section 10.4] services can be instantiated by more than one receiving activity through messages that can be received in a statically unpredictable order: the first incoming message triggers creation of a service instance which subsequent messages are delivered to.

nating different Web services to obtain a new, customised service. In practice, and briefly, WSDL [11] is a W3C standard that permits to express the functionalities offered and required by web services by defining, akin to 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 orchestrations, 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 values of their parameters are 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 as follows: `<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 as follows: `<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 with 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 transactions. 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 performing 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 suffice 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 [18] 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. In this way, a message can be delivered to the correct instance on the basis of the values associated with the correlation variables, independently of any routing mechanism.

3.2 An overview of COWS

COWS [23,37] 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, asyn-

Table 1 COWS syntax

Expressions: ϵ, ϵ', ...
Values: v, v', ...
Names: n, m,...
Partners: p, p', ...
Operations: o, o', ...
Variables: x, y, ...
Variables/Values: w, w', ...
Variables/Names: u, u', ...

Services:

s ::=		
	$u \bullet u'!\bar{\epsilon}$	(invoke)
$\mid$	g	(receive-guarded choice)
$\mid$	$s \mid s$	(parallel composition)
$\mid$	$[u]\, s$	(delimitation)
$\mid$	$* s$	(replication)

Receive-guarded choice:

g ::=		
	$\mathbf{0}$	(empty)
$\mid$	$p \bullet o?\bar{w}.s$	(receive)
$\mid$	$g + g$	(choice)

chronous communication, polyadic synchronisation, 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 [37]) 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.

Services are structured activities built from basic activities, i.e. the empty activity $\mathbf{0}$, the invoke activity $\_ \bullet \_!\_$ and the receive activity $\_ \bullet \_?\_$, 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. We will omit trailing occurrences of $\mathbf{0}$, writing, for example, $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$. Moreover, we will write $[\langle u_1, \ldots, u_n \rangle]\, s$ in place of $[u_1] \ldots [u_n]\, s$ and use $I \triangleq s$ to assign a name I to the term s.

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 $p \bullet o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. An invoke $p \bullet o!\langle \epsilon_1, \ldots, \epsilon_n \rangle$ can proceed as soon as all expression arguments ϵ_1, ..., ϵ_n can be successfully evaluated to values. A receive $p \bullet o?\langle w_1, \ldots, w_n \rangle.s$ offers an invocable operation o along a given partner name p; 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 received names cannot form the endpoints used to receive further invocations. In other words, endpoints of receive activities are identified statically[2] 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 [37] for further details) is allowed to progress (namely, execution of this receive takes precedence over that of 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 Sect. 5).

Delimitation is the only binding construct: $[u]\, s$ binds the element u in the scope s. It is used for two different purposes:

[2] This choice has been influenced by the current (web) service technologies where endpoints of receive activities are statically determined. We refer the interested reader to [37] for further motivations, and comparisons with respect to the existing literature, of this and other choices regarding the design of COWS.

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.

4 Blind-date conversations in WS-BPEL

We present here a (web) service capable of arranging matches of 4-player online (card) games, such as *burraco* or *canasta*.

It is worth noticing that the blind-date conversation strategy works well also with a number of players not fixed a priori, but for the sake of simplicity and to better highlight the specificities of the proposed strategy, we have fixed the number of players to four and we have kept the functionality to the smallest working scenario. Later on, in Sect. 6, we provide an extended variant of this service that supports games with: a dynamic number of players, players' anticipated disconnection from a match, and merging of matches. This two-step presentation permits to gradually introduce technicalities and distinctive features thus making understanding easier. It also points out that with the blind-date joining strategy we can build really complex services.

By using the presented service, in order 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, as a consequence of their request to play, players do not either know if a new match is created or instead they join an already existing match. Therefore, as required by the basic principles of the blind-date conversation strategy presented in Sect. 2, the arrangement of tables is *completely transparent* to players.

Figure 2 shows a graphical representation of the WS-BPEL process, called `TableManager`, implementing the service described above. We report below the code of the process where, to simplify 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.

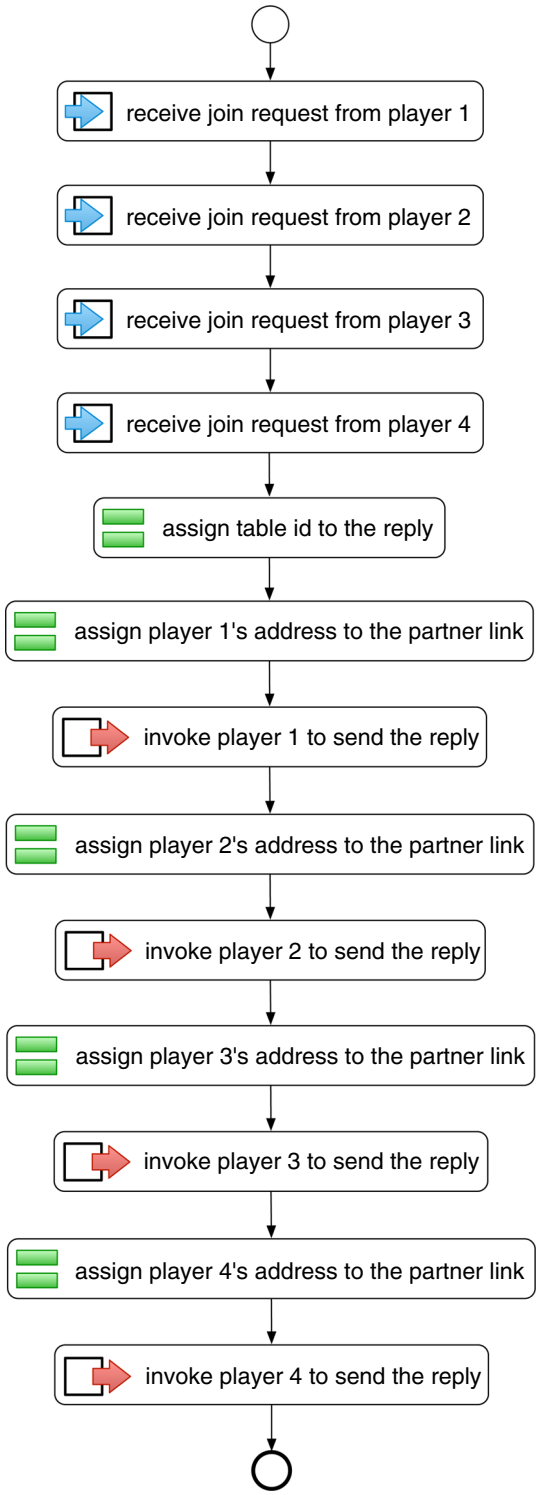

Fig. 2 Graphical representation of the WS- BPEL process `TableManager`

```
<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. The used message style[3] is *document*; thus, every message is formed by a single part, called `payload`, that contains all data carried in the message. 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 in 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 a 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 with the process as follows:

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

<prop:propertyAlias messageType="this:request"
               part="payload"
               propertyName="this:gameNameProperty">
```

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

```
  <prop:query queryLanguage=...>
    //this:RequestElement/this:gameName
  </prop:query>
</prop:propertyAlias>
```

A <property> specifies an element of a correlation set and relies on one (or more) <propertyAlias> 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 an instance, in order to complete the card-table for the new match. 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 we do not model as it is out of the scope of this work).

Using the* WS-BPEL *process. This WS- BPEL process can be used via the Web interface at http://reggae.dsi.unifi.it/blinddatejoining/, or by downloading source and binary code from https://sourceforge.net/projects/blinddatejoining/.

The process is configured to be deployed in a WS-BPEL engine Apache ODE [1]. Indeed, we have equipped the process with the corresponding WSDL document and deployment descriptor file, 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 the above expression as a random function.

To facilitate the use of the WS- BPEL process, we have developed a sort of testing environment consisting of a few Java classes implementing the service clients. Such classes rely on the artefacts automatically generated by JAX-WS [15] 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 [36]; a screenshot of the application is shown in Fig. 3. Notably, players created in a given browser session could be assigned to tables together with players created in other sessions.

5 Semantics mechanisms underlying the blind-date conversation joining

In this section, to clarify the behaviour at run-time 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).

To facilitate the comprehension of the COWS specification, and its comparison with the WS- BPEL process of the previous section, in Table 2 we sketch the translation of WS- BPEL communication activities in COWS constructs. Notation $p_{partnerLinkName}$ refers to the endpoint reference stored in the corresponding WS- BPEL partner link, p_{cb} represents a (fresh) endpoint reference for the callback communication, while $u_{partnerLinkName}$ refers to a partner link that may be either already initialised or not. We refer the interested reader to [39] for further details on the correspondence between WS- BPEL activities and COWS constructs. Our translation is quite intuitive and direct, which makes us confident that the original semantics is obeyed, although we did not prove that in general the semantics of the COWS specification resulting from a translation 'conforms', in some formal way, to that of the original WS- BPEL process.

Communication activities, i.e. invokes and receives, are translated in different ways depending on the interaction pattern they are part of. Indeed, WS- BPEL supports three different interaction patterns among clients and servers: one-way, (synchronous) request–response and asynchronous request–response patterns. One-way pattern is the simplest interaction pattern: the server providing the operation performs the receive activity, whereas the client performs the invoke activity. Our translation shows that one-way <receive> and <invoke> activities are directly supported in COWS. We use u_{plc} to indicate the fact that the provider's address can be already known at design time (when $u_{plc} = p$) or be discovered at run-time (when $u_{plc} = x$). A synchronous request–response interaction corresponds to a pair of one-way interactions in COWS, i.e. the request and the callback. Thus, COWS forces the client to send the partner name p_{cb} to be used by the provider for sending the reply back to the

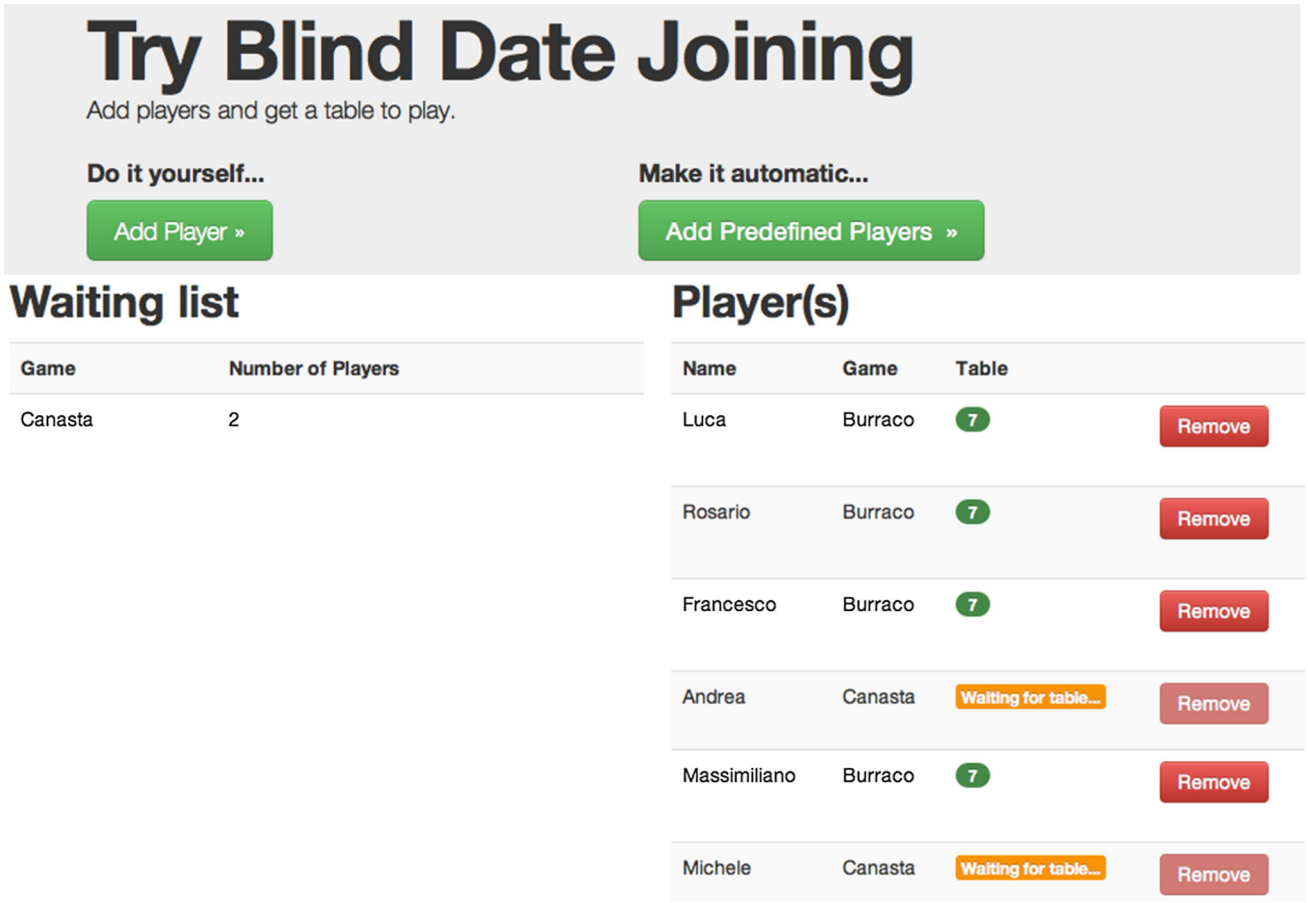

Fig. 3 A screenshot of the Web application facilitating the use of the process `TableManager`

Table 2 Translation of WS- BPEL communication activities in COWS constructs

WS-BPEL	COWS
One-way interactions	
`<invoke partnerLink="plc" operation="op" inputVariable="x" />`	$u_{plc} \bullet op!\bar{x}$
`<receive partnerLink="pls" operation="op" variable="y" />`	$p_{pls} \bullet op?\bar{y}$
Request-response interactions	
`<invoke partnerlink="plc" operation="op" inputVariable="x1" outputVariable="x2" />`	$[p_{cb}]\,(u_{plc} \bullet op!(p_{cb}, \bar{x}_1) \mid p_{cb} \bullet op?\bar{x}_2)$
`<receive ... /> ... <reply partnerlink="pls" operation="op" variable="y2" />`	$p_{pls} \bullet op?(z, \bar{y}_1) \cdots z \bullet op!\bar{y}_2$
Asynchronous request-response interactions	
`<invoke partnerlink="plc" operation="op1" variable="x1" /> ... <receive partnerLink="plc" operation="op2" variable="x2" />`	$u_{plc} \bullet op_1!(p_{plc}, \bar{x}_1) \cdots p_{plc} \bullet op_2?\bar{x}_2$
`<receive partnerLink="pls" operation="op1" variable="y1" /> ... <invoke partnerlink="pls" operation="op2" variable="y2" />`	$p_{pls} \bullet op_1?(u_c, \bar{y}_1) \cdots u_c \bullet op_2!\bar{y}_2$

client. On the other hand, to be able to handle such a request, the server providing the operation is ready to receive also such a partner name (stored in the variable z). An asynchronous request–response is rendered through a partner link connecting two one-way interactions.

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

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

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 $x_{player i}$, with $i = 1..4$. The endpoint *manager* • *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 requests, the players are required to provide only the kind of game, stored in x_{game}, and their partner names, stored in $x_{player i}$, that they will then use to receive the table identifier. Player requests are received through receive activities of the form $\mathit{manager} \bullet \mathit{join}?\langle x_{game}, x_{player i}\rangle$, 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 an identifier will be then communicated to each player by means of four invoke activities $x_{player i} \bullet \mathit{start}!\langle \mathit{tableId}\rangle$. 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 $\mathit{manager} \bullet \mathit{join}$.

Consider now the following system

$$\mathit{Luca} \mid \mathit{Rosario} \mid \mathit{Francesco} \mid \ldots \mid \mathit{TableManagerProcess}$$

where the players are defined as follows

$$
\begin{aligned}
\mathit{Luca} &\triangleq \mathit{manager} \bullet \mathit{join}!\langle \mathit{burraco}, p_L\rangle \\
&\quad \mid [x_{id}] \, p_L \bullet \mathit{start}?\langle x_{id}\rangle . \langle \text{rest of } \mathit{Luca}\rangle \\
\mathit{Rosario} &\triangleq \mathit{manager} \bullet \mathit{join}!\langle \mathit{canasta}, p_R\rangle \\
&\quad \mid [x_{id}] \, p_R \bullet \mathit{start}?\langle x_{id}\rangle . \langle \text{rest of } \mathit{Rosario}\rangle \\
\mathit{Francesco} &\triangleq \mathit{manager} \bullet \mathit{join}!\langle \mathit{burraco}, p_F\rangle \\
&\quad \mid [x_{id}] \, p_F \bullet \mathit{start}?\langle x_{id}\rangle . \langle \text{rest of } \mathit{Francesco}\rangle \\
&\ldots
\end{aligned}
$$

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

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

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

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

When *Francesco* invokes *TableManagerProcess*, the process definition and the first created instance, being both able to receive the same message $\langle \mathit{burraco}, p_F\rangle$ along the endpoint *manager* • *join*, compete for the request $\mathit{manager} \bullet \mathit{join}!\langle \mathit{burraco}, p_F\rangle$. COWS's (prioritised) 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

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

where *Francesco* joined 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 by communicating them the table identifier:

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

Now, the players of table *tableId* (including *Luca* and *Francesco*) can start playing, while *Rosario* keeps waiting for other *canasta* players.

Besides clarifying the run-time effects and the underlying mechanisms of the blind-date joining strategy, the COWS specification above could be also exploited to analyse relevant properties of the `TableManager` process, in a way similar to the study reported in [16]. Although analysis is out of this paper's scope, here we give a taste of the reasonings that can be carried out on the considered scenario thanks to the COWS semantics.

More specifically, the COWS specification enables the application of two main analysis techniques. The first one relies on the temporal logic SocL and the COWS model checker CMC [14] specifically devised for verifying properties relevant for the service-oriented domain over COWS specifications. This approach permits expressing and checking functional properties of services. Thus, for example, we could verify that the `TableManager` process is *available* (i.e. it is always ready to accept join requests) and *responsive* (i.e. whenever it receives a request, it eventually provides a single response, assuming that there are at least other three clients willing to play the same game).

The second analysis technique relies on the type system introduced in [24] for checking *confidentiality* properties of services. In this way, for example, it is possible to ensure that a player cannot pass the table identifier, received from the `TableManager` process, to another user, thus preventing the latter user from impersonating the former one.

6 A more realistic case study

We present here the extended version of the process shown in Sect. 4 that permits a more complex handling of card game matches. The aim of this case study is twofold. On the one hand, it permits demonstrating the effectiveness of the blind-date strategy on a more realistic scenario than the one discussed in the previous sections, which has indeed a purely illustrative purpose. On the other hand, it provides a complete account of the lifecycle of conversations using the blind-date strategy; in fact, although we mainly focus on the joining phase, this case study also considers the progress, merging and completion phases of conversations.

The new service can determine the number of players at run-time, so managed games are not limited to those with a specific number of players. It also allows players to give up a match whenever they want, also before its termination. Furthermore, when the minimum number of participants is not reached and no further request arrives within a given amount of time, the enhanced service exploits merging of conversations to give to waiting players the possibility of starting a match. To implement the new capabilities we also resort to the XSLT [29] language, in order to conveniently manage the players list.

In addition to the partner link `tableManager`, used to receive and respond to players requests, the process uses the partner link `merge` to send merging requests to other conversations. The merging of a conversation is implemented by sending to the manager service itself a separate join request for each participant in the conversation that has to be merged. In this way, waiting players can join an existing match where a place has been previously freed by a player disconnection. If currently existing matches have no free place, a new match is created. As prescribed by the blind-date join strategy, the client is not informed if he will join a new match or an existing one. Notably, in case of low turnout of new players and low disconnection rate, the same player can be repeatedly reassigned to a new match. Thus, for reducing the rate of recreation of the same incomplete conversation, the timeout triggering the conversation merging must be set to an appropriate value, which depends on the end-user connectivity profiles of the considered game. It is worth noticing that, to properly handle conversation joining and merging requests, we need to use two separate partner links of the same type, although both kinds of requests are sent to the same operation `join` provided by the same service. This is due to the fact that a WS- BPEL process cannot play both the receiver and sender roles on the same partner link.

The enhanced process uses two correlation sets: `GameCorrelationSet`, for managing the joining of players to a match, and `TableIDCorrelationSet`, for managing player logouts and match termination signals. We need two correlation sets because logouts and termination signals are allowed only on a specific table identifier. On the other hand, join requests are not bound to a specific table, so the game name is used as correlation identifier.

The process also uses eight variables:

- a variable for each operation, i.e. `PlayerRequest` for operation `join`, `PlayersReply` for `start`, `LogoutMsg` for `logout`, and `GameOverMsg` for `gameOver`;
- four variables to support conversation management, i.e. `Players` (which is a sort of array) to save the information about players that have joined the current conversation, `RequiredPlayers` to save the minimum number of players required to start a match of the selected game, `MaxNumberOfPlayers` to save the maximum number of players admitted by the selected game, and `NumberOfPlayers` to store the current number of players.

Figure 4 shows a graphical representation of the extended `TableManager` process implementing the service described above. The figure outlines the structure of the process, while the skeleton description below reports further details, e.g. conditions and assignment expressions. The corresponding WS- BPEL code[4] can then be easily derived from this description. The COWS specification can be derived in a similar way. In this case, the skeleton description is refined by replacing the WS- BPEL communication constructs by the corresponding COWS communication activities (as shown in Table 2) and rendering the standard imperative constructs (assignment, sequence, if-then-else, etc.) in COWS as shown in [23].

```
receive tableManager join PlayerRequest;
Players[1] = PlayerRequest.address;
RequiredPlayers =
    getNumberOfRequiredPlayers(PlayerRequest.gameName);
MaxNumberOfPlayers =
    getMaxNumberOfPlayers(PlayerRequest.gameName);
NumberOfPlayers = 1;
while (NumberOfPlayers < RequiredPlayers) {
   pick (receive tableManager join PlayerRequest) {
      NumberOfPlayers = NumberOfPlayers + 1;
      Players[NumberOfPlayers] = PlayerRequest.address
   } timeout(300S) {
        while (NumberOfPlayers > 0) {
           PlayerRequest.address =
               Players[NumberOfPlayers];
           NumberOfPlayers = NumberOfPlayers - 1;
           invoke merge join PlayerRequest
        }
        exit
   }
}
PlayersReply.tableID = getTableID();
NumberOfPlayers=0;
while (NumberOfPlayers < RequiredPlayers) {
   NumberOfPlayers = NumberOfPlayers + 1;
   tableManager = Players[NumberOfPlayers];
   invoke tableManager start PlayersReply
}
while (true) {
   if (NumberOfPlayers < MaxNumberOfPlayers) {
      pick (receive tableManager join PlayerRequest) {
            NumberOfPlayers = NumberOfPlayers + 1;
            Players[NumberOfPlayers] =
                PlayerRequest.address;
            tableManager = PlayerRequest.address;
            invoke tableManager start PlayersReply
      } (receive tableManager logout LogoutMsg) {
         NumberOfPlayers = NumberOfPlayers - 1;
         Players =
             removePlayer(Players,LogoutMsg.address)
      } (receive tableManager gameOver GameOverMsg) {
         exit
      }
   } else {
      pick (receive tableManager logout LogoutMsg) {
         NumberOfPlayers = NumberOfPlayers - 1;
         Players =
             removePlayer(Players,LogoutMsg.address)
      } (receive tableManager gameOver GameOverMsg) {
         exit
      }
   }
}
```

Like its simpler version, the service initiates the conversation using the first receive. This receive will initialise also the correlation set `GameCorrelationSet` with the requested game name. Then, it will store the information about the first player and calculate the minimum number of players for starting a match of the requested game by using the custom XPath

[4] For the sake of readability, the WS- BPEL code is relegated to Appendix, while source and binary code can be downloaded from https://sourceforge.net/projects/blinddatejoining/.

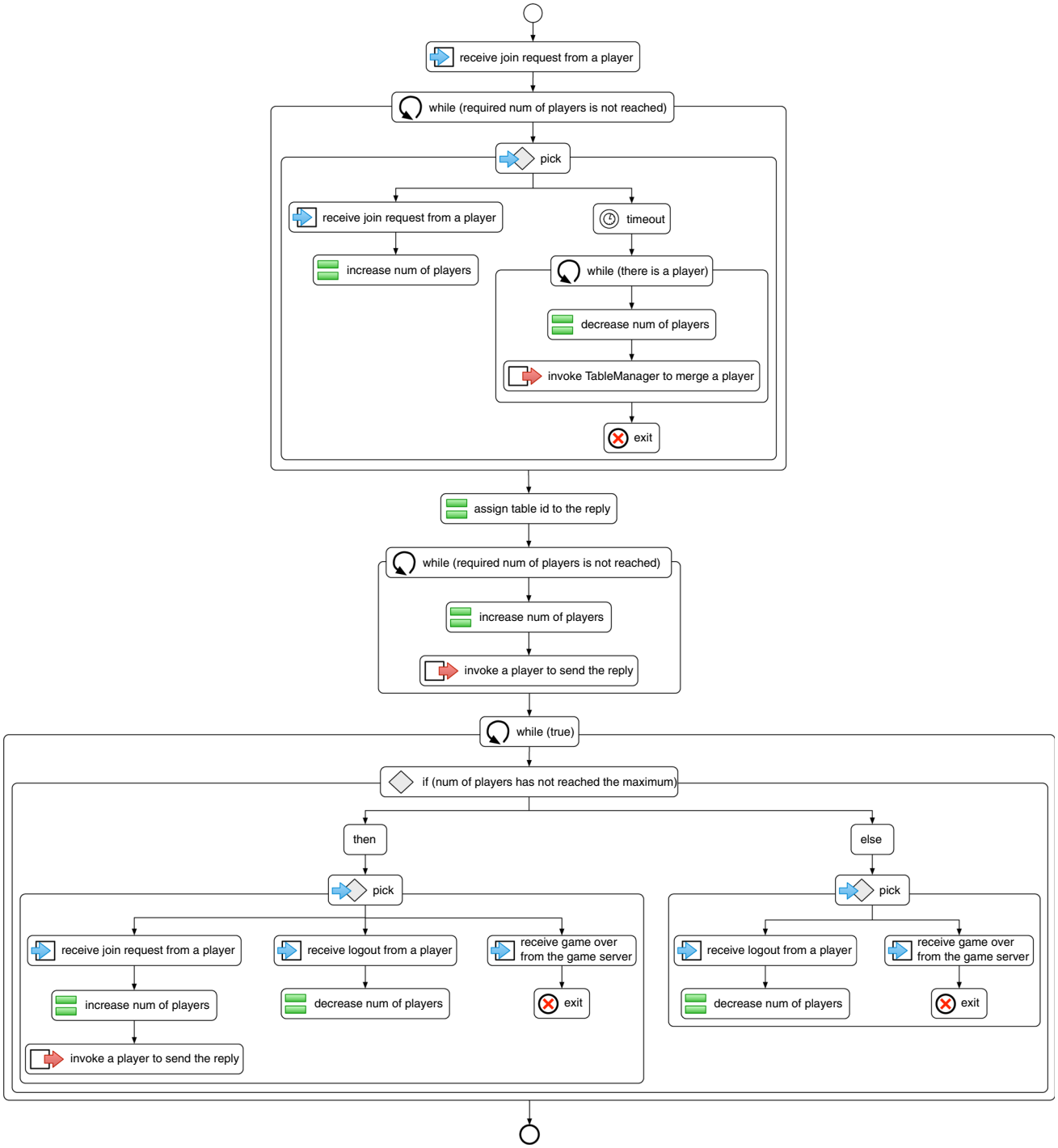

Fig. 4 Graphical representation of the extended `TableManager` process

function `getNumberOfRequiredPlayers`. This function requires the game name as input and returns, indeed, the minimum number of players needed by the game. Similarly, the service calculates the maximum number of players for the given game. Afterwards, the service will wait for the right amount of players to be achieved. For each received join request, the following assignment is executed:

```
<assign>
 ...
 <copy>
  <from>doXslTransform("AddToPlayers.xsl",
                       $Players.payload)</from>
  <to>$Players.payload</to>
 </copy>
 <copy>
  <from>$PlayerRequest.payload/tm:address</from>
  <to>$Players.payload/tm:
```

```
            player[$NumberOfPlayers]/tm:address</to>
  </copy>
 </assign>
```

The XSLT transformation `AddToPlayers` is used to add an empty entry to the players list, then the information about the player is added to this list by using an XPath expression that selects the previously added empty entry in the list and fills it with the address of the player.

The process uses a `<pick>` activity to associate a time-out with the reception of a join request. In this way, if the required number of players is not yet achieved and a join request is not received within a predefined amount of time, the current conversation is merged with other ones. The time-out is implemented by setting the desired amount of time through a `<onAlarm>` element within the `<pick>` activity. The merge mechanism is then carried out by resending the request to join a match on behalf of all the players involved in the conversation. To accomplish these operations the information inside the `Players` variable is used to build the request messages that will be sent through the partner link `merge`. In this way, if other conversations on the same topic are available, due to the prioritised communication mechanism of WS- BPEL, they will accept the players as new participants; otherwise, new conversations will be created to accommodate the players.

Once the desired amount of players is reached, the table identifier is generated by means of the custom XPath function `getTableID`. Then, this information is sent to each player participating in the conversation via a loop that selects every address in the players list, sets that address as destination of the partner link `tableManager`, and sends the table identifier. It is worth noticing that this loop initialises the correlation set `TableIDCorrelationSet` by using the join strategy. This means that the first invocation initialises the correlation set with the table identifier, then the others simply use this initialisation.

At this point, as in the previous variant, the players can start playing the requested game by using the table identifier and by interacting with a dedicated game server.

Afterwards, the process enters in a loop that cycles until the match termination signal from the game server is received. During this phase, the service can behave in two different ways depending on whether the maximum number of players is achieved or not. In the former case, the process can only receive disconnection requests and match termination signals. In the latter case, instead, the process can additionally receive joining requests. If a disconnection request is received, the process decrements the number of players and removes the requesting player from the `Players` list using the XSLT transformation `RemovePlayer`. If a match termination signal is received, instead, the process simply uses the `<exit>` activity to terminate the conversation (i.e. the WS- BPEL service instance). Finally, if a joining request is received, the information about the requesting player is stored, the number of players is incremented, and the table identifier is communicated to the new player.

7 Related work

Service interaction in SOC. 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 [34] could be used as the correlation-based orchestration language and SOCK [19] as the formalism, where the former is 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 engines. Instead, COWS has been selected because of its strict correspondence with WS-BPEL. At the same time, it is a core calculus with just a few constructs, which makes it more suitable than WS- BPEL to reason on applications' behaviour.

In the SOC literature (see, for example, [7] for a review), 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 [42] where, however, only interactions among different instances of a single service are taken into account. This makes this approach not suitable for the blind-date join strategy, as it deals with multiparty scenarios. Another correlation-based formalism, besides COWS and SOCK mentioned above, is the calculus *Corr* [28], which is a sort of value-passing CCS [31] without restriction and enriched with constructs for expressing services and their instances. We have discarded *Corr* for illustrating the run-time effects of the blind-date strategy, because it is engine dependent, as it has been specifically designed to capture behaviours related to correlation aspects in the ActiveBPEL engine, and because its definition is more complex as it provides much more constructs than COWS. 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 [27]. This work proposes an implementation of a correlation-based primitive allowing messages to be broadcasted to more than one service conversation. This form of interaction, however, differs from that required to express blind-date joining scenarios,

which need a form of interaction allowing multiple parties to send messages to the same service conversation.

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 [32]) which is implicitly instantiated when calling a service: it binds caller and callee and is used for their future conversation. Although this approach does not have a direct correspondence with the technology underling SOC, its abstraction level has proved convenient for reasoning about SOC applications. Indeed, session-based conversation can be regulated by the so-called *session types* (see [21] for a survey). They can statically guarantee a number of desirable properties, such as communication safety, progress, and predictability. Therefore, an important group of calculi for modelling and proving properties of services is based on the explicit notion of interaction session. Most of such work has been devoted to studying *dyadic* sessions, i.e. interaction sessions between only two participants. In particular, in this strand of work we would like to mention the Global and End-point calculi [9], SCC [2], SSCC [22], and CaSPiS [3]. However, these approaches can only manage conversations between two parties, while, as explained before, the blind-date join strategy requires multiparty conversations.

Another body of work focussed on a more general form of sessions, called *multiparty* sessions/conversations, which is closer to the notion of conversation that we have considered in this paper. The multiparty session approach proposed in [20] permits expressing a conversation by means of channels shared among the participants. However, the conversation is created through a single synchronisation among all participants, whose number is fixed at design time. The only way to allow a new partner to participate to a session is by means of the delegation mechanism, which, however, ousts the delegating partner from the session. This differs from our notion of blind-date joining, where once a conversation has been created the participants can asynchronously join as the number of participants to a conversation can change at run-time. The μse language [4] 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 previously created conversations. Instead, when relying on correlation, the conversation merging is automatically performed for each correlated request. This simplifies and makes more transparent the conversation join, as needed for implementing a blind-date approach. Another formalism dealing with multiparty interactions is the Conversation Calculus [8,41]. It uses the conversation-based mechanism for inter-session communication, which permits to progressively accommodate and dismiss participants from the same conversation. This is realised by means of named containers for processes, called conversation contexts. However, processes in unrelated conversations cannot interact directly. Moreover, conversation joining is not transparent to a new participant, because he has to know the name of the conversation. In our approach, instead, 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 fulfil 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.

Online games implementation. Online game playing is very common nowadays, and it is supported by a large number of heterogeneous hardware platforms. Every game that supports online multiplayer gaming has to provide a strategy that permits players to engage an online session with other players. This strategy must tackle two problems:

- find other players;
- manage the online sessions.

Players are usually found by using a matchmaking server, or selecting them from pre-existent matches or on invitation. The matchmaking server is used when a player wants to join or create a game session but does not know the other players (as, for example, in [12,26]). Thus, a pool of servers is used to process the requests of the interested players at any time. In order to improve game experience and ease the work of software providers, cloud technologies are used for instantiating a number of servers adequate to the actual requests for the game [5]. The selection from a list of servers is a simplification of the previous mechanism because in this case the server only retains the list of the actual joinable game sessions, but it is the user that enacts the matchmaking phase (see, for example, [33]). Instead, the creation of a game session through invitation is done when a player decides to create herself a session and to invite the other players that she wants to play with. Those techniques can be used before the beginning of a session or while a game session is taking place (almost all games support this option). Anyway, the choice of a strategy for allowing players to engage an online session with other players heavily depends on the kind of game. As witnessed by the application to the online game case study in Sects. 4–6, the blind-date strategy can be conveniently used in this domain to accommodate online matches for all

such games where user interactions and preferences are not required in the matchmaking phase. In particular, the use of high-level correlation-based constructs should relieve the programmer of dealing explicitly with matches creation and management.

In order to minimise the waiting time before playing, the search of other players often has a timeout that permits the system to stop the searching phase and replace the unavailable human players with artificial players [13]. By exploiting this mechanism, the session can quickly start and can be populated with human players when they arrive. We adopted a similar solution based on timeouts in the extended variant of the table manager process presented in Sect. 6.

When players of a session are found, a game session must be created. The most used strategies for this are client–server or peer-to-peer. The first mechanism uses a server to host the game session, so all the players connect to the server in order to play. As said before, cloud technologies can be used to scale the number of servers relatively to the client requests. The peer-to-peer strategy instead does not need a dedicated server to host the game session because a player is chosen (with some kind of policy) to host the game (see, for example, [6]). In our case study we followed the client–server approach, which was the more natural choice given the SOC languages, WS- BPEL and COWS, we selected for implementing and formalising the blind-date joining strategy.

Lastly, online game services [30,38,40] often allow players to login on a server in order to store statistics about their performances. This information will be used to optimise the matchmaking phase. It is important to note that the login usually does not cause a player to join a game session. Therefore, we have not taken into account this feature of online games, because it is not relevant for our study on joining mechanisms.

8 Concluding remarks

We have illustrated the blind-date conversation joining, a strategy allowing a participant to join a conversation without need to know information about the conversation itself, except for the endpoint of the service provider. We have shown how this strategy can be implemented by using the WS- BPEL correlation sets mechanism and formalised by using the process calculus COWS. Moreover, we have developed a case study to show how our strategy can be used in a realistic scenario that involves an online games provider that arranges matches of card games. The scenario is implemented using WS- BPEL and executed on Apache ODE. Despite the XML markup used by WS- BPEL, the blind-date joining strategy permits to have smooth and clean implementations. We have also equipped the WS- BPEL process with a 'testing environment' in order to facilitate its use.

Many other features could be added without much effort to our WS- BPEL process, so that they would make the case study even more realistic. For example, a player should be able to play concurrently in more than one match. Moreover, here we propose a possible merge strategy, but many other versions of merge can be adopted. For example, a centralised solution could exploit the table manager as a forwarder for the player messages, while a decentralised one could exploit the tables without enough players as forwarders. Finally, also the dedicated game server can use different strategies to deal with player disconnections. For example, if the number of players of a match decreases under the minimum threshold, the match can be suspended until a new player joins it; once this happens, the match can either resume or restart. Alternatively, disconnected players could be replaced by computer-controlled ones, until new human players join the conversation.

A main limitation for the practical use of the blind-date joining mechanism is the need of high-level correlation-based constructs, such as those provided by orchestration languages like WS- BPEL. Indeed, as shown in this paper, such constructs are really convenient for enacting the blind-date strategy, but they are not natively available in mainstream programming languages (Java, C++, etc.). Thus, a programmer willing to use such languages should also take charge of implementing all the mechanisms enabling the blind-date strategy, i.e. shared input variables, pattern matching, and priority among receive activities. His task would be relieved if an API providing the required functionalities is available. Design and development of such an API is left as a future work.

Finally, the scope of the study carried out in this paper is limited to the presentation, also via formal modelling, of the blind-date joining. However, the COWS specification introduced here lends itself to the application of many analysis techniques for regulating the correlation mechanism in order to guarantee desirable properties. We have mentioned some of these techniques in Sect. 5 and leave the study of further techniques as a future work.

Acknowledgements We thank the anonymous referees for their useful comments.

Appendix

We report here the WS- BPEL code of the (enhanced) service introduced in Sect. 6.

```
<process name="TableManager" ... >
<partnerLinks>
  <partnerLink name="tableManager"
    myRole="table" partnerRole="player"
    partnerLinkType="tm:table" ... />
  <partnerLink name="merge"
    partnerRole="table"
```

```
    partnerLinkType="tm:table" ... />
</partnerLinks>
<variables>
  <variable messageType="tableManager:request"
            name="PlayerRequest"/>
  <variable messageType="tableManager:reply"
            name="PlayersReply"/>
  <variable messageType="tableManager:logout"
            name="LogoutMsg"/>
  <variable messageType="tableManager:gameOver"
            name="GameOverMsg"/>
  <variable messageType="tableManager:playersList"
            name="Players"/>
  <variable type="xs:int" name="RequiredPlayers" />
  <variable type="xs:int" name="MaxNumberOfPlayers" />
  <variable type="xs:int" name="NumberOfPlayers" />
</variables>
<correlationSets>
  <correlationSet name="GameCorrelationSet"
                  properties="tm:gameNameProperty"/>
  <correlationSet name="TableIDCorrelationSet"
                  properties="tm:tableIDProperty"/>
</correlationSets>
<sequence>
  <receive partnerLink="tableManager" operation="join"
           variable="PlayerRequest"
           createInstance="yes" >
    <correlations>
      <correlation initiate="yes"
                   set="GameCorrelationSet"/>
    </correlations>
  </receive>
 <assign>
  ...
  <copy>
   <from>$PlayerRequest.payload/tm:address</from>
   <to>$Players.payload/tm:player[1]/tm:address</to>
  </copy>
  <copy ... >
   <from>t:getNumberOfPlayers($PlayerRequest.payload/
   tm:gameName)</from>
   <to>$RequiredPlayers</to>
  </copy>
  <copy ...>
   <from>
    t:getMaxNumberOfPlayers($PlayerRequest.payload/
    tm:gameName)
   </from>
   <to>$MaxNumberOfPlayers</to>
  </copy>
  <copy>
    <from>1</from>
    <to>$NumberOfPlayers</to>
  </copy>
 </assign>
 <while>
  <condition>
  $NumberOfPlayers < $RequiredPlayers
  </condition>
  <pick>
   <onMessage partnerLink="tableManager"
              operation="join"
              variable="PlayerRequest" ... >
    <correlations>
     <correlation initiate="no"
                  set="GameCorrelationSet"/>
    </correlations>
    <assign>
     <copy>
      <from>$NumberOfPlayers + 1</from>
      <to variable="NumberOfPlayers"/>
     </copy>
     <copy>
      <from>
      doXslTransform("AddToPlayers.xsl",
                     $Players.payload)
      </from>
      <to>$Players.payload</to>
     </copy>
     <copy>
      <from>$PlayerRequest.payload/tm:address</from>
      <to>
      $Players.payload/tm:player[$NumberOfPlayers]/
      tm:address
      </to>
     </copy>
    </assign>
   </onMessage>
   <onAlarm>
    <for>'PT300S'</for>
    <sequence>
     <while>
      <condition> $NumberOfPlayers > 0 </condition>
      <sequence>
       <assign>
       <copy>
        <from>
        $Players.payload/tm:player[$NumberOfPlayers]/
        tm:address
        </from>
        <to>$PlayerRequest.payload/tm:address</to>
       </copy>
       <copy>
        <from>$NumberOfPlayers - 1</from>
        <to variable="NumberOfPlayers"/>
       </copy>
       </assign>
       <invoke operation="join" partnerLink="merge"
               inputVariable="PlayerRequest"/>
      </sequence>
     </while>
     <exit/>
    </sequence>
   </onAlarm>
  </pick>
 </while>
 <assign>
  ...
  <copy ...>
   <from>table:getTableID()</from>
   <to>$PlayersReply.payload/tableManager:tableID</to>
  </copy>
  <copy>
   <from>
   $Players.payload/tm:player[1]/tm:address
   </from>
   <to partnerLink="tableManager"/>
  </copy>
  <copy>
   <from>0</from>
   <to>$NumberOfPlayers</to>
  </copy>
 </assign>
 <while>
  <condition>
  $NumberOfPlayers < $RequiredPlayers
  </condition>
  <sequence>
  <assign>
   <copy>
    <from>$NumberOfPlayers + 1</from>
    <to variable="NumberOfPlayers"/>
   </copy>
   <copy>
    <from>
    $Players.payload/tm:player[$NumberOfPlayers]/
    tm:address
    </from>
    <to partnerLink="tableManager"/>
   </copy>
  </assign>
  <invoke inputVariable="PlayersReply"
          operation="start"
          partnerLink="tableManager">
   <correlations>
    <correlation initiate="join"
                 set="TableIDCorrelationSet"/>
   </correlations>
  </invoke>
```

```
 </sequence>
</while>
<while>
 <condition> true() </condition>
 <if>
  <condition>
  $NumberOfPlayers < $MaxNumberOfPlayers
  </condition>
  <pick>
   <onMessage operation="join"
              partnerLink="tableManager"
              variable="PlayerRequest" ...>
    <correlations>
     <correlation initiate="no"
                  set="GameCorrelationSet"/>
    </correlations>
    <sequence>
    <assign>
     <copy>
      <from>$NumberOfPlayers + 1</from>
      <to variable="NumberOfPlayers"/>
     </copy>
     <copy>
      <from>
      doXslTransform("AddToPlayers.xsl",
                     $Players.payload)
      </from>
      <to>$Players.payload</to>
     </copy>
     <copy>
      <from>$PlayerRequest.payload/tm:address</from>
      <to>
      $Players.payload/tm:player[$NumberOfPlayers]/
      tm:address
      </to>
     </copy>
     <copy>
      <from>
      $PlayerRequest.payload/tableManager:address
      </from>
      <to partnerLink="tableManager"/>
     </copy>
     </assign>
     <invoke operation="start"
             partnerLink="tableManager"
             inputVariable="PlayersReply" />
    </sequence>
   </onMessage>
   <onMessage operation="logout"
              partnerLink="tableManager"
              variable="LogoutMsg" ...>
    <correlations>
     <correlation initiate="no"
                  set="TableIDCorrelationSet"/>
    </correlations>
     <assign>
      <copy>
       <from>$NumberOfPlayers - 1</from>
       <to variable="NumberOfPlayers"/>
      </copy>
      <copy>
       <from>
       doXslTransform("RemovePlayer.xsl",
                      $Players.payload,"RmPlayer",
                      $LogoutMsg.payload/tm:address)
       </from>
       <to>$Players.payload</to>
      </copy>
     </assign>
   </onMessage>

   <onMessage operation="gameOver"
              partnerLink="tableManager"
              variable="GameOverMsg" ...>
    <correlations>
     <correlation initiate="no"
                  set="TableIDCorrelationSet"/>
    </correlations>
    <exit/>
   </onMessage>
  </pick>
 </if>
</while>
</sequence>
</process>
```

References

1. Apache Software Foundation: Apache ODE 1.3.6 (2013) http://ode.apache.org/
2. Boreale M, Bruni R, Caires L, De Nicola R, Lanese I, Loreti M, Martins F, Montanari U, Ravara A, Sangiorgi D, Vasconcelos V, Zavattaro G (2006) SCC: a service centered calculus. WS-FM, LNCS, vol 4184. Springer, Berlin, pp 38–57
3. Boreale M, Bruni R, De Nicola R, Loreti M (2015) Caspis: a calculus of sessions, pipelines and services. Math Struct Comput Sci 25(3):666–709
4. Bruni R, Lanese I, Melgratti H, Tuosto E (2008) Multiparty sessions in SOC. In: Lea D, Zavattaro G (eds) Coordination, LNCS, vol 5052. Springer, Berlin, pp 67–82
5. Bruno J (2013) Games on Xbox one—better with Xbox live compute. http://news.xbox.com/2013/10/15/xbox-one-cloud/
6. Bungie: Halo (2016) http://www.halowaypoint.com/
7. Caires L, De Nicola R, Pugliese R, Vasconcelos VT, Zavattaro G (2011) Core calculi for service-oriented computing. In: Rigorous software engineering for service-oriented systems—results of the SENSORIA project on software engineering for service-oriented computing, LNCS, vol 6582. Springer, Berlin, pp 153–188
8. Caires L, Vieira H (2010) Conversation types. Theor Comput Sci 411(51–52):4399–4440
9. Carbone M, Honda K, Yoshida N (2012) Structured communication-centered programming for web services. ACM Trans Program Lang Syst 34(2):8:1–8:78

10. Cesari L, Pugliese R, Tiezzi F (2013) Blind-date Conversation Joining. In: EPTCS, WWV, vol 123, pp 3–18
11. Chinnici R, Moreau J, Ryman A, Weerawarana S (2007) Web services description language (WSDL) 2.0. Tech. rep., W3C. https://www.w3.org/TR/wsdl20/
12. EA: Battlefield (2016) http://www.battlefield.com/
13. Epic software: gears of war (2016) http://gearsofwar.xbox.com/
14. Fantechi A, Gnesi S, Lapadula A, Mazzanti F, Pugliese R, Tiezzi F (2012) A logical verification methodology for service-oriented computing. ACM Trans Softw Eng Methodol 21(3):16:1–16:46
15. GlassFish community: JAX-WS (2016) https://jax-ws.java.net/
16. Gnesi S, Pugliese R, Tiezzi F (2011) The sensoria approach applied to the finance case study. In: Rigorous software engineering for service-oriented systems—results of the SENSORIA project on software engineering for service-oriented computing, LNCS, vol 6582. Springer, Berlin, pp 698–718
17. Groupon (2016) http://www.groupon.com/
18. Gudgin M, Hadley M, Rogers T (2006) Web services addressing 1.0—Core. Tech. rep., W3C
19. Guidi C, Lucchi R, Gorrieri R, Busi N, Zavattaro G (2006) SOCK: a calculus for service oriented computing. In: ICSOC, LNCS, vol 4294. Springer, Berlin, pp 327–338
20. Honda K, Yoshida N, Carbone M (2016) Multiparty asynchronous session types. J ACM 63(1):9
21. Hüttel H, Lanese I, Vasconcelos VT, Caires L, Carbone M, Deniélou P, Mostrous D, Padovani L, Ravara A, Tuosto E, Vieira HT, Zavattaro G (2016) Foundations of session types and behavioural contracts. ACM Comput Surv 49(1):3
22. Lanese I, Martins F, Ravara A, Vasconcelos V (2007) Disciplining orchestration and conversation in service-oriented computing. In: SEFM. IEEE Computer Society Press, pp 305–314
23. Lapadula A, Pugliese R, Tiezzi F (2007) A calculus for orchestration of web services. In: ESOP, LNCS, vol 4421. Springer, pp 33–47
24. Lapadula A, Pugliese R, Tiezzi F (2007) Regulating data exchange in service oriented applications. In: FSEN, LNCS, vol 4767. Springer, pp 223–239
25. Lapadula A, Pugliese R, Tiezzi F (2012) Using formal methods to develop WS-BPEL applications. Sci Comput Program 77(3):189–213
26. MAG interactive: Ruzzle (2016) http://www.ruzzle-game.com/
27. Mauro J, Gabbrielli M, Guidi C, Montesi F (2011) An efficient management of correlation sets with broadcast. In: COORDINATION, LNCS, vol 6721. Springer, Berlin, pp 80–94
28. Melgratti H, Roldán C (2012) On correlation sets and correlation exceptions in ActiveBPEL. In: TGC, LNCS, vol 7173. Springer, Berlin, pp 212–226
29. Michael Kay (2007) XSL transformations (XSLT) version 2.0. Tech. rep., W3C
30. Microsoft: Xbox live (2016) http://www.xbox.com/
31. Milner R (1989) Communication and concurrency. Prentice-Hall, Englewood Cliffs
32. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, I and II. Inf Comput 100(1):1-40–41-77
33. Mojang: Minecraft (2016) https://minecraft.net/
34. Montesi F, Guidi C, Lucchi R, Zavattaro G (2007) JOLIE: a Java orchestration language interpreter engine. In: MTCoord, ENTCS, vol 181. Elsevier, pp 19–33
35. OASIS WSBPEL TC: Web services business process execution language version 2.0. Tech. rep., OASIS (2007) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
36. Play framework: play framework 2.5 (2016) http://www.playframework.com/
37. Pugliese R, Tiezzi F (2012) A calculus for orchestration of web services. J Appl Log 10(1):2–31
38. Sony: playstation network (2016) http://playstation.com/
39. Tiezzi F (2009) Specification and analysis of service-oriented applications. PhD thesis in computer science, Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze. http://rap.dsi.unifi.it/cows
40. Valve: steam (2016) http://store.steampowered.com/
41. Vieira H, Caires L, Seco JC (2008) The conversation calculus: a model of service-oriented computation. In: ESOP, LNCS, vol 4960. Springer, Berlin, pp 269–283
42. Viroli M (2004) Towards a formal foundation to orchestration languages. In: WS-FM, ENTCS, vol 105. Elsevier, pp 51–71