Composition-oriented Service Discovery

Antonio Brogi, Sara Corfini, Razvan Popescu

Department of Computer Science
University of Pisa, Italy

Abstract. Service discovery and service aggregation are two crucial issues in the emerging area of Service-oriented Computing (SoC). We propose a new technique for the discovery of (Web) services that accounts for the need of composing several services to satisfy a client query. The proposed algorithm makes use of OWL-S ontologies, and explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result. When no full match is possible, the algorithm features a flexible matching by returning partial matches and by suggesting additional inputs that would produce a full match.

1 Introduction

Service-oriented Computing (SoC) [10] is emerging as a new, promising computing paradigm that centres on the notion of *service* as the fundamental element for developing software applications. According to [10], services are self-describing components that should support a rapid and low-cost composition of distributed applications. Services are offered by service providers, which procure service implementations and maintenance, and supply service descriptions. Service descriptions are used to advertise service capabilities, behaviour, and quality, and should provide the basis for the discovery, binding, and composition of services. Services possess the ability of engaging other services in order to complete complex transactions, like checking credit, ordering products, or procurement. The platform-neutral nature of services creates the opportunity for building composite services by composing existing elementary or complex services, possibly offered by different service providers [14].

The Web service model includes three component roles — clients, providers and registries — where providers advertise their services to registries, and clients query registries to discover services. In this scenario, two prominent issues involved in the development of next generation distributed software applications can be roughly synthesised as:

(1) discovering available services that can be exploited to build a needed application, and
(2) suitably aggregating such services to achieve the desired result.

Currently, the universally accepted core standard employed for Web service discovery is the Universal Description & Discovery Interface (UDDI [11]). UDDI

supports the definition of service registries in the style of yellow pages, but unfortunately it features only keyword-based matches that often give poor performance.

Given the pivotal importance of service discovery for SoC, several attempts to improve the quality of UDDI discovery are currently being pursued. One of the major efforts in this direction is promoted by the World Wide Web Consortium (W3C) which strongly advocates the introduction of semantic information in the description of Web services [12]. Indeed, currently service description are expressed by means of the Web Services Description Language (WSDL [13]), by declaring a set of message formats and their direction (incoming/outgoing). The resulting description is purely syntactic, very much in the style of Interface Description Languages (IDLs) in component-based software engineering.

The W3C proposes a semantic-based description of Web services, based on the use of OWL-S (formerly DAML-S) ontologies [8], where each service is provided with an *advertisement* containing three descriptions: *service profile* ("what the service does"), *service model* ("how the service works"), and *service grounding* ("how to access the service"). The process of Web service discovery — often referred to as service matchmaking — then takes a query specifying inputs and outputs (IOs) of the desired service as well as a service registry consisting of (service) advertisements, and returns as output a list of matched services.

In this paper we present a new algorithm for the composition-oriented discovery of Web services. The algorithm — called SAM (for Service Aggregation Matchmaking) — can be used to match queries with service registries making use of OWL-S ontologies. SAM extends a matchmaking algorithm proposed by Bansal and Vidal in [3] by featuring a more flexible matching and, more importantly, by accounting for service compositions. Indeed, queries that cannot be satisfied by a single service might be frequently satisfied by composing several services. An immediate example of this is a client wishing to plan its holidays by booking flight tickets as well as hotel accommodation while taking into account various parameters such as weather, season prices, special offers, and so on.

The main features of the proposed algorithm can be summarised as follows:

- *Flexible matching.* The proposed algorithm (SAM) features a more flexible matching with respect to. [3] as:
  - SAM performs a fine-grained matching at the level of atomic processes, or sub-services (rather than at the level of entire services as in [3]).
  - Rather than returning only full matches (when a single service can fully satisfy the client request by itself), SAM also returns (when no full match is possible) a list of *partial* matches. A partial match is a (composition of) sub-service(s) that can provide only some of the outputs requested by the client. It is important to stress that a partial match can be a valuable answer for the client, which may have over-specified its query or may decide to use the selected services even if its query will be only partially satisfied.
  - When no full match is possible, SAM — besides returning partial matches — is also capable to suggest to the client additional inputs that would suffice to get a full match.

– *Composition-oriented matching.* More importantly, SAM is the first algorithm (at the best of our knowledge) to provide a composition-oriented matchmaking based on semantic descriptions of queries and services by taking into account service process models.
  - When no single service can satisfy the client query, SAM checks whether there are service compositions that can satisfy the query, possibly including multiple executions of services as well.
  - When SAM finds a match, it explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result.

The rest of the paper is organised as follows. Section 2 is devoted to introduce OWL-S ontologies for service discovery, while in Section 3 we describe the new algorithm for the composition-oriented discovery of services. Finally, we draw some concluding remarks in Section 4.

2 Web Service Discovery Using OWL-S Ontologies

As the use of UDDI for service discovery often leads to inaccurate matches, increasing attention is being devoted to semantics-based techniques to improve the quality of the matchmaking process. The best known approaches use DAML-S/OWL-S [8] ontologies. OWL-S is an ontology for describing Web services and it is written in DAML+OIL. The root of the ontology is represented by the generic class Service which has three subclasses: service profile ("what the service does"), service model ("how the service works") and service grounding ("how to access the service").

The service profile provides a high-level description of a service and it consists of three types of information: a human readable description of the service, the functionalities provided by the service, and some functional attributes. Service functionalities are represented by listing the inputs required as well as the outputs produced by the service, and functional attributes specify additional information about the service such as what guarantees of response time or accuracy it provides, or the cost of the service.

The service model has a process model subclass which provides a view of a Web service in terms of process compositions. OWL-S defines three types of processes: atomic, simple and composite. An atomic process is executed in a single step (from the point of view of the client of the service). It can not be decomposed further and it has an associated grounding. Only atomic processes are allowed to have associated inputs and outputs (IOs) and they are the only processes that can be directly invoked by the client. A simple process is similar to an atomic one but it can not be invoked directly and it does not have an associated grounding. It is a simplified and abstract view of a composite process. Finally, a composite process consists of other processes, the composition being made with the following control constructs: `split`, `sequence`, `unordered`, `split+join`, `choice`, `if-then-else`, `iterate` and `repeat-until/while`.

The first matchmaking algorithms based on DAML-S/OWL-S ontologies (e.g., [9]) use the service profile. The matching based on the service profile (similar somehow to matching two black boxes) allows to match a service request asking for two outputs o_1 *and* o_2 with a service advertisement that provides *either* o_1 *or* o_2 but *not necessarily both* o_1 *and* o_2 (e.g., a `choice` process). Indeed, in order to clearly specify the behaviour of such service one would have to provide two service profiles corresponding to the two alternatives. As one may note this would lead to advertising a large number of profiles, even for non trivial services. Moreover, analysing Web services only through their service profile (i.e., their IOs), severely affects the process of discovery of service aggregations that satisfy a request. Indeed, the service profile does not describe the internal behaviour of services and hence it does not provide valuable information needed for composing services.

Bansal and Vidal present in [2,3] an improvement of the matchmaking process by using an algorithm based on the OWL-S process model. Their algorithm takes as input a query specifying the desired IOs as well as a repository of OWL-S Web services and returns one of the following degrees of match: `exact` (e.g., client asks for "DVD", provider replies with "Digital Versatile Disk"), `plug-in` (e.g., client asks for "British Music DVDs", provider replies with "Music DVDs"), `subsumes` (e.g., client asks for "Music DVDs", provider replies with "British Music DVDs"), or `failed` (e.g., client asks for "DVD", provider replies with "MC"). The algorithm takes into account the process model trees of the advertisements as well as the ontological relations between matched IOs. According to [9], a service request matches a service advertisement if the request provides all the inputs (possibly more) needed by the advertisement while the advertisement generates all the outputs (possibly more) needed by the requester. The algorithm of Bansal and Vidal stores OWL-S service advertisements as trees corresponding to their process models. Composite processes correspond to intermediary nodes while atomic processes are represented as leaves. The root of the process model corresponds to the root of the tree. The matchmaking algorithm begins at the root of the advertisement tree and recursively visits all its subtrees finishing at the leaves. For each node (e.g., `sequence`, `choice` and so on) a corresponding matching algorithm that verifies the compatibility between its IOs and the IOs of the query is employed. For example, in the case of a `sequence` process, if the outputs requested by the query can be satisfied by all its children collectively then we have a success, otherwise a failure. In the case of a `choice` process we get a success or a failure depending on whether there exists at least one child able to provide by itself all the outputs desired by the query. A detailed description of the matching algorithms corresponding to several composite processes can be found in [2].

Two of the main limitations of existing matchmaking algorithms are single service discovery and single service execution. Indeed, existing algorithms look (inside a repository) for a *single* service capable to fulfil the request. For example, let us consider a registry containing two services: Electronics_Store (Figure 1) and Online_Bank (Figure 2). The first sells electronic items like notebooks or digital cameras. The second is able to create virtual credit cards; a client

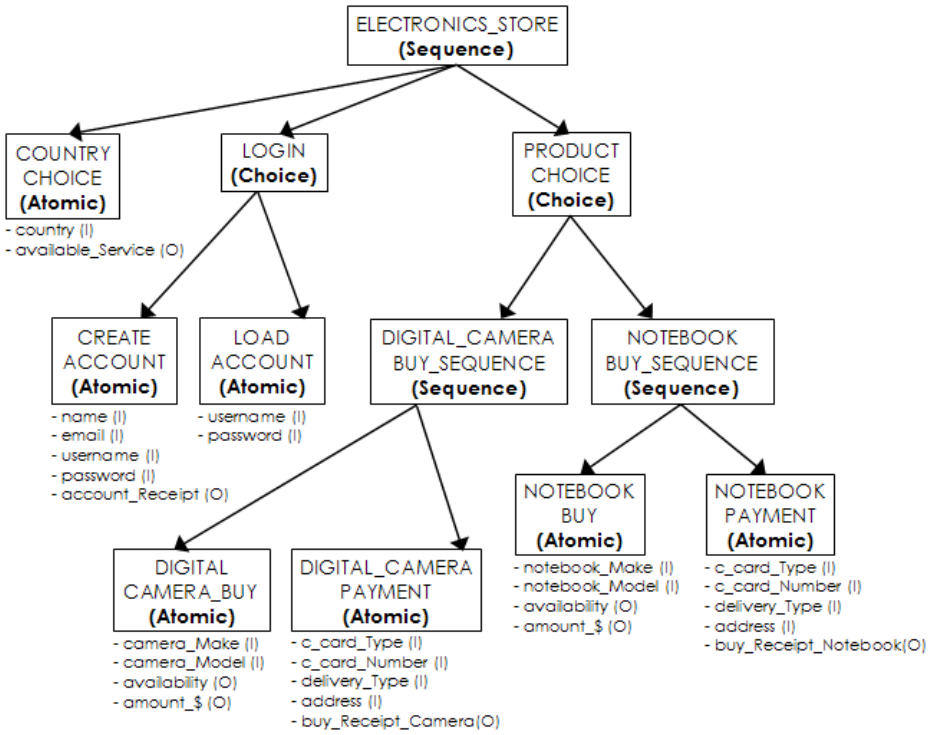

Fig. 1. Process model of an Electronics Store Service.

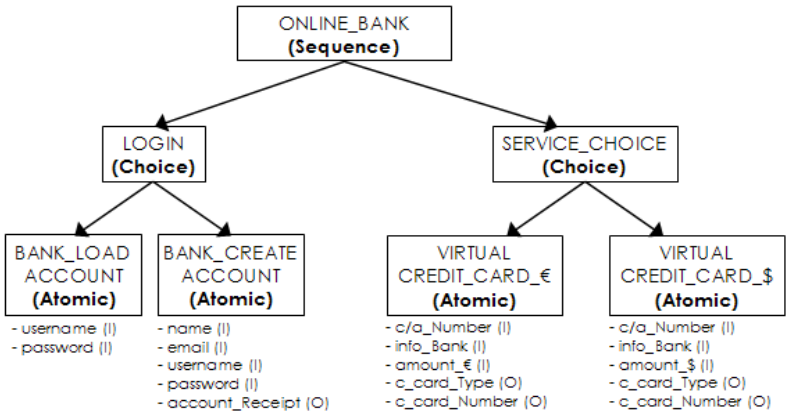

Fig. 2. Process model of an Online Bank Service.

obtains a credit card number and a credit card type through a bank transfer. We suppose that all concepts contained in the OWL-S advertisements are defined in a shared ontology. Consider now the query specifying:

– inputs: `username`, `password`, `country`, `notebook_Model`, `notebook_Make`, `c/a_Number`, `info_Bank`, `delivery_Type` and `address`, and
– output: `buy_Receipt_Notebook`.

Existing matchmaking algorithms give a `failed` match because in the registry there is no service able to fulfil the request by itself. On the other hand, we can observe that the `c_card_Type` and `c_card_Number` needed as inputs by the `Notebook_Payment` atomic process of the ELECTRONICS_STORE service are not provided by the query but they can be obtained by executing the ONLINE_BANK service. In other words, while the first service cannot satisfy the query, a suitable composition of the two services can.

A second limitation of existing matchmaking algorithms is that they do not consider multiple executions of services. Consider for instance the query specifying:

– inputs: `username`, `password`, `country`, `camera_Model`, `camera_Make`, `notebook_Model`, `notebook_Make`, `delivery_Type`, `address`, `c_card_Type`, and `c_card_Number`, and
– outputs: `buy_Receipt_Notebook` and `buy_Receipt_Camera`.

We observe that while existing algorithms return a `failed` match for this query, it could be satisfied by executing *twice* the ELECTRONICS_STORE service. Indeed, the `Digital_Camera_Buy_Sequence` and `Notebook_Buy_Sequence` composite processes of the first service are children of a `choice` process and hence they cannot be both executed in a single run.

The following section is devoted to present a composition-oriented algorithm for service discovery that overcomes the above described limitations.

3 Service Aggregation Matchmaking (SAM)

The goal of the SAM algorithm is to determine whether a query can be satisfied by a (composition of) service(s), advertised in an OWL-S registry. SAM starts with a preliminary phase during which it builds a tree for each process model stored in the registry, as described by Bansal and Vidal in [3] and as we have summarised in the previous section. The SAM algorithm consists of two main parts (which will be described in the next two subsections):

1. Construction of a graph representing the dependencies among atomic processes of the services in the registry;
2. Analysis of such dependency graph to determine a service composition capable to satisfy the query (or part of it, when no service composition can fully satisfy the query).

3.1 Construction of the Dependency Graph

The graph produced during the matchmaking phase is a directed graph. It has two node types: *data* node and *process* node, the former corresponding to data

(inputs or outputs) and the latter to matched processes. In SAM the match regards exclusively atomic processes, the only processes that can be directly invoked by the client. An atomic process matches if and only if:

- either all its inputs are available because they are part of the query or because they are returned as outputs by other previously matched atomic processes,
- or at least one of its outputs is part of the query or it is an input for some previously matched atomic process.

When the algorithm finds a new matched atomic process, it creates a corresponding process node and adds it to the graph. For each process node inserted in the graph, the algorithm creates and inserts (if not already present) a data node for each input and output of such process. If a data node is an input for the process node, SAM inserts a directed edge from the former to the latter. Similarly, if a data node is an output of the process node, the algorithm inserts an edge from the latter to the former. There are also two types of edges between process nodes, called *sequencing* and *excluding* constraints. If a process node P_2 is a successor of another process node P_1 as both are children of a `sequence` construct, the algorithm inserts an edge $P_1 \rightarrow P_2$ (*sequencing* constraint). We also say that P_1 is a predecessor of P_2. SAM introduces a bidirectional edge $P_1 \leftrightarrow P_2$ in the graph for each pair of process nodes (P_1, P_2) that are children of a `choice` process (*excluding* constraint).

Initially, the graph contains only the data nodes corresponding to the inputs and the outputs of the query. The matchmaking phase cycles over the registry until no more process nodes can be added to the dependency graph. The matchmaking is implemented by a recursive function `Match`, invoked over each service. The `Match` function starts its execution at the root of the advertisement tree and it is recursively invoked over children nodes. The execution finishes at leaf nodes, where `Match` verifies the compatibility between the inputs and the outputs of the corresponding atomic process and the data nodes currently present in the graph. According to the OWL-S specification [8], we assume that an output O_i is *compatible* with an input I_j if and only if either O_i and I_j represent the same concept (*exact match*), or O_i represents a sub-concept of I_j ("O_i plugs-in I_j", or equivalently "I_j subsumes O_i"). `Match` deals with different types of OWL-S nodes (`sequence`, `choice` and so on). For atomic nodes for example, `Match` checks whether the corresponding atomic process is already contained in the graph. If this is not the case, `Match` verifies the compatibility between the inputs and the outputs of the atomic node and the data nodes currently contained in the graph. If all its inputs or at least one of its outputs are contained (w.r.t. compatibility) in the graph then the atomic process is considered to be matched and added to the graph. `Match` then creates a new process node, new data nodes and all needed edges and constraints, and inserts them in the dependency graph. In the case of a `sequence` node, `Match` verifies if the corresponding sub-tree contains at least one matched atomic process. If so, all (matched and unmatched) atomic processes contained in the sub-tree are inserted in the dependency graph. For a `choice` node it verifies if it has at least one matched atomic process. In

this case, differently from a `sequence` node, only the matched atomic process children are added to the graph.

The behaviour of the `Match` function is summarised by the following pseudo-code, where I_p and O_p denote the inputs and the outputs of P, respectively. Let also PREV_p be the set of atomic processes which must be executed before P, and CHOICE_p be the set of atomic processes which can be executed only if P is not executed.

```
MATCH(ServiceRegistry SR, Query Q, Graph G)
    repeat
        forall service S in SR do MATCH (Root(S), Q, G);
    until no process node is added to G;


MATCH(AtomicProcess P, Query Q, Graph G)
    if (P∉ G) then
        if (I_p ∈ G ∨ O_p ∩ G ≠ ∅) then
            Add P to G;
            forall outputs O in O_p do
                if (O ∉ G) then Add O to G;
                Add (P, O) to G;
            forall inputs I in I_p do
                if (I ∉ G) then Add I to G;
                Add (I, P) to G;
            forall predecessors PR in PREV_p do
                if (PR ∈ G) then Add (PR, P) to G;
            forall choice processes PC in CHOICE_p do
                if (PC ∈ G) then Add (P, PC) ∧ (PC, P) to G;


MATCH(SequenceProcess SP, Query Q, Graph G)
    forall child C in SP.children do
        MATCH(C, Q, G);
    if (at least one process node is added to G) then
        Add all atomic processes of SubTree(SP) to G;


MATCH(ChoiceProcess CP, Query Q, Graph G)
    forall child C in CP.children do MATCH(C, Q, G);
```

3.2 Analysis of the Dependency Graph

The second phase of the algorithm consists of analysing the dependency graph constructed during the first phase. This second phase consists of five steps, described next.

Step 1. **Reachability of query outputs**. The dependency graph includes a data node for each query input and output, regardless of whether or not these data have been matched during the first phase. SAM hence first checks whether there are query output nodes in the graph G that do not have incoming edges from process nodes. Indeed, such disconnected query outputs can not be produced as no service in the registry can generate them. If there are disconnected query outputs in the graph, the client has to choose whether the matchmaking process should nevertheless continue (by discarding such outputs from the query) or abort. In the latter case SAM terminates with a FAILURE. In the former case SAM removes the disconnected query outputs and continues with Step 2.

Step 2. **Yellow Colouring**. In this step SAM identifies — by colouring them in yellow — all processes which *may* be useful for generating the query outputs. Initially all nodes in the graph are white. The white colour is used to denote all process and data nodes that do not have yet proved to be useful for satisfying the query. SAM first colours in yellow all the query outputs. It then recursively paints in yellow all process and data nodes that are white and that have an outgoing edge leading to a yellow node. Note that excluding constraints are not taken into account here (i.e., the yellow paint does not spread over excluding constraints). The process of painting in yellow finishes when there is no other node that can be coloured. At the end of this step all yellow process nodes correspond to processes that might have to be executed in order to generate the query outputs. Dually, yellow data nodes correspond to data that "might be useful as input"/"might be generated as output" to/by yellow processes in order to generate the query outputs. All nodes that are still white at the end of this phase are not needed for fulfilling the request (and could be removed from the graph). One may note that more nodes than necessary may have been painted. The algorithm then continues with Step 3.

Step 3. **Red&Black Colouring**. The goal of this step is to identify — by painting them in red — the processes which contribute to generate the query outputs and which can be actually executed given the query inputs are provided. To describe this step it is convenient to introduce the notion of FIRABLE process. *A process node P is* FIRABLE *in a graph G if P is yellow and all its input data nodes are red and if there are predecessor processes linked through sequencing constraints then there is at least one such predecessor process node coloured in red.* The algorithm first paints in red all data nodes corresponding to query inputs. While there is at least one yellow query output node and at least one FIRABLE process, the algorithm selects a FIRABLE process for execution. If there are several FIRABLE processes linked through excluding constraints then SAM non-deterministically chooses one such FIRABLE process node and paints it in red. Every non-deterministic choice corresponds to a "fork" (split) into several instances. After painting a process in red, all its output data nodes are coloured in red and all the process nodes linked to it by excluding constraints are inhibited by painting them in black. (We do so as for example, by colouring in red a `Pay_with_Credit_Card` process we should inhibit another `Pay_with_Cash` process linked to it by an excluding constraint.) When painting in red a process node, the algorithm adds it to a PROCESS_SEQUENCE list initially empty. Each instance of this step finishes either with a SUCCESS if all query outputs became red, or with a FAILURE if there are no more FIRABLE processes but there is still at least one yellow query output. It is important to note that if there are several FIRABLE processes linked through excluding constraints then the non deterministic CHOOSE operator splits the current execution of this step into a number of instances equal to the number of FIRABLE processes, each such instance corresponding to painting in red the respective process node and further on its outputs as well as to inhibiting the processes linked to it by excluding constraints. As a result of this step we shall obtain a set of triples <SUCCESS/FAILURE, COLOURED

GRAPH G, PROCESS_SEQUENCE>. Next, SAM continues with Step 4.

Step 4. **Analysis of Triples**. The algorithm further checks whether there is at least one tuple <SUCCESS, G, PROCESS_SEQUENCE>. If so, it returns to the client an ordered list of all tuples T_i where T_i = {<SUCCESS, G_i, PROCESS_SEQUENCE$_i$>}. Such list can be ordered by taking into account client's preferences (expressed together with the query). Such preferences can include minimal number of matched services, minimal PROCESS_SEQUENCE length and so on.

Now, in the case that all the triples generated by Step 3 are FAILUREs, SAM checks whether there exists a set of FAILUREs that together are able to generate all outputs requested by the query. If so, the request can be satisfied by simply considering one of the possible sequences of the FAILUREs in this set. It is important to note again that the choice of such set is made with respect to client's preferences. If such a set exists, the process finishes by returning to the client a sequence obtained by the concatenation of all PROCESS_SEQUENCEs corresponding to the considered FAILUREs in the set. In this case we have a SUCCESS obtained from the aggregation of a set of FAILUREs.

If instead there is no such set of FAILUREs that can collectively satisfy the query it means that there are query outputs that remain yellow in all graphs obtained at the end of Step 3. The algorithm then computes the intersection of the sets of all such unsatisfiable query outputs for all FAILUREs. Next, similarly to the previous case, it considers a set of FAILUREs able to collectively satisfy the producible outputs (i.e., the query outputs less the unsatisfiable ones). The algorithm further asks the client whether it wishes more information with respect to what is needed to completely satisfy the request. This information consists of the additional inputs that are needed in order to be able to unlock and to execute other processes so as to fully satisfy the request. If the client agrees then SAM continues with Step 5. Otherwise it terminates.

Step 5. **Individuating Additional Inputs**. During this last step the algorithm looks for additional inputs that need to be provided in order to have further FIRABLE processes that help generating the unsatisfiable query outputs. Hence, for each FAILURE and for each unsatisfied output, SAM looks for yellow process nodes that generate this output. The set of additional inputs needed for producing this output in the respective FAILURE comes from considering all yellow input data nodes of these processes and recursively all yellow input data nodes of the processes that should be executed before them due to sequencing constraints[1].

The following pseudo-code summarises the analysis of the dependency graph described so far.

Let $G = (N, E)$, where $E = E_{dp} \bigcup E_{pd} \bigcup E_{sc} \bigcup E_{cc}$, E_{pd} = links from process to data nodes of the form (P, D), E_{dp} = links from data to process nodes of the form (D, P), as well as sequencing constraints: $(P', P) \in E_{sc}$, and excluding constraints: $(P', P) \in E_{cc}$. Let I_p and O_p denote the inputs and the outputs of a process P respectively, and let $Q = \{I_Q, O_Q\}$ denote the query. Let also O_F be the set of data output nodes that are red in failure F.

[1] If two or more processes generate the same output then they are taken as alternatives. The same happens for processes with more than one yellow predecessors.

$U = \{D \in O_Q \mid \nexists P : (P, D) \in E\}$; //Step 1 → Reachability of Query Outputs
if $U \neq \emptyset$ **then**
    Query client whether to go ahead ignoring U;
    if client says yes **then** $O_Q = O_Q \setminus U$;
    else
        return(*"Query cannot be satisfied"*);
Paint in white all X s.t. $X \in N$;

Paint in yellow all D s.t. $D \in O_Q$; //Step 2 → Yellow Colouring
while $\exists X, Y : (X, Y) \in E_{dp} \cup E_{pd} \cup E_{sc} \wedge Xwhite \wedge Yyellow$ **do** Paint X in yellow;

Initialise ProcessSequence; //Step 3 →Red & Black Colouring
Paint in red all D s.t. $D \in I_Q \wedge D$ yellow;
while ($Firable(G) \neq \emptyset \wedge \exists D \in O_Q$: D yellow) **do**
    if $\exists P' : (P'firable \wedge \forall P" \in Firable(G) : (P', P") \notin E_{cc})$ **then** $P = P'$;
    else P = choose(Firable(G));
    Paint P in red and add P to ProcessSequence;
    $\forall D : D$ yellow $\wedge (P, D) \in E_{pd}$: paint D in red;
    $\forall P' : P'$ yellow $\wedge (P, P') \in E_{cc}$: paint P' in black;
if $\exists D : D$ yellow $\wedge D \in O_Q$ **then** failure;
else success;

if there exists at least one success **then** //Step 4 →Analysis of Triples
    return *an ordered list of (successful) results*;
else
    if $\exists$ a set S of failures s.t. $\forall D \in O_Q \ \exists F \in S : D \in O_F$ **then**
        return *a concatenation of the ProcessSequences of all graphs in* S;
    else
        $NonProducibleOutputs = \{D | D \in O_Q \wedge \forall$ failure $F : D$ yellow in $F\}$;
        $ProducibleOutputs = O_Q \setminus NonProducibleOutputs$;
        $S = \{F \mid F$ is a failure$\} \wedge \forall D \in ProducibleOutputs \ \exists F \in S : D \in O_F$;
        AddI = $\emptyset$;
        Query client whether it wants info on additional inputs;
        if client says yes **then**
            forall failure F **do** // Step 5 → Individuating Additional Inputs
                forall $D \in NonProducibleOutputs$ **do**
                    $P = Q \in G \mid D \in O_Q$;
                    $AI = \{D \mid (D \in I_P \wedge D$ yellow$)\}$;
                    while $(\exists P' \mid (P', P) \in E_{sc})$ **do**
                        $AI = AI \cup \{D \mid (D \in I_{P'} \wedge D$ yellow$)\}$;
                        $P = P'$;
                    $AddI = AddI \cup$ *"Add inputs needed for"* $+ D +$ *"in"* $+ F +$ *":"* $+ AI$;
        return (*a concatenation of the ProcessSequences of all graphs in* S, AddI);

3.3 An Example

We shall present next an example that illustrates the behaviour of SAM. Let us consider a registry of OWL-S advertisements containing only the two services – Electronics_Store and Online_Bank – described in Section 2.
Consider now the query specifying:

- inputs: `username`, `password`, `country`, `camera_Model`, `camera_Make`, `notebook_Model`, `notebook_Make`, `c/a_Number`, `info_Bank`, `delivery_Type` and `address`, and
- outputs: `buy_Receipt_Camera` and `buy_Receipt_Notebook`.

One may note that Bansal and Vidal's algorithm [2,3] would return a failure because there is no service in the registry able to satisfy the query by itself. More precisely, the Electronics_Store service requires `c_card_Type` and `c_card_Number` as inputs in order to be able to provide `buy_Receipt_Camera` and `buy_Receipt_Notebook`. Yet these inputs are not provided with the request

but are to be obtained by executing the ONLINE_BANK service. Bansal and Vidal's algorithm fails as it is unable to find such relation between IOs of distinct services.

In its first phase SAM constructs the dependency graph (the graph in Figure 3 without colours) as the result of the matching process between the query and the registry of services advertisements. One may note the exclusion constraints between the `Load_Account` and `Create_Account` atomic processes, between the `Bank_Load_Account` and `Bank_Create_Account`, as well as between `Notebook_Buy`, `Digital_Camera_Buy`, `Notebook_Payment` and `Digital_Camera_Payment` atomic processes.

During the second phase SAM continues as follows:

1. During the first step SAM checks whether there are disconnected query outputs in the graph, yet in this example all query outputs are produced by at least an atomic process.
2. SAM continues next with the second step ("Yellow Colouring") during which it paints in yellow all data nodes corresponding to the query outputs and then it recursively paints in yellow all process and data nodes linked to other yellow nodes. At the end of this step all data nodes are painted in yellow with the exception of `available_Service`, `account_Receipt` and `availability` concepts.
3. During step three ("Red&Black Colouring") SAM starts by painting in red all yellow data nodes corresponding to the query inputs. At this point the only FIRABLE processes are `Country_Choice` and `Bank_Load_Account` as all their data inputs are red and they have no predecessors. Let us consider that SAM chooses to execute the `Country_Choice` process. By doing so, it paints it in red and it adds it to the PROCESS_SEQUENCE list. Moreover, the `Load_Account` process becomes FIRABLE as its (unique) predecessor is now red. By further assuming that SAM selects the `Load_Account` process for execution, it paints it in red and then it inhibits the `Create_Account` process by painting it in black. In our case, the algorithm continues until there are no more FIRABLE processes available. One may note that there is a moment in which both `Digital_Camera_Buy` and `Notebook_Buy` are FIRABLE. At that point SAM splits the execution in two instances: the first one paints in red the `Digital_Camera_Buy` process and it paints in black the `Notebook_Buy` and `Notebook_Payment` processes, while the second paints in red the `Notebook_Buy` process and it paints in black the `Digital_Camera_Buy` and `Digital_Camera_Payment` processes. At the end of step three of the algorithm both instances return a FAILURE as they were unable to generate (i.e., to paint in red) all the requested outputs — the first one produces the `buy_Receipt_Camera` but not the `buy_Receipt_Notebook` while the second one produces the `buy_Receipt_Notebook` but not the `buy_Receipt_Camera`. The PROCESS_SEQUENCE list resulting from the first instance is hence [`Country_Choice`, `Load_Account`, `Bank_Load_Account`, `Digital_Camera_Buy`, `Virtual_Credit_Card_$`, `Digital_Camera_Payment`][2]. The second instance pr-

[2] The corresponding coloured graph is shown in Figure 3.

oduces the following PROCESS_SEQUENCE list: [Country_Choice, Load_Account, Bank_Load_Account, Notebook_Buy, Virtual_Credit_Card_$, Notebook_Payment].

4. SAM continues with step four of the algorithm ("Analysis of Triples"). Due to the fact that both instances of the previous step return a FAILURE, it checks whether their union is able to generate all the requested outputs. Consequently, SAM obtains a SUCCESS from the aggregation of the PROCESS_SEQUENCEs corresponding to the two FAILUREs. SAM finishes by returning to the client the following PROCESS_SEQUENCE: [[Country_Choice, Load_Account, Bank_Load_Account, Digital_Camera_Buy, Virtual_Credit_Card_$, Digital_Camera_Payment], [Country_Choice, Load_Account, Bank_Load_Account, Digital_Notebook_Buy, Virtual_Credit_Card_$, Notebook_Payment]].
5. For our scenario, the last step of the algorithm is not executed as the request has been fulfilled.

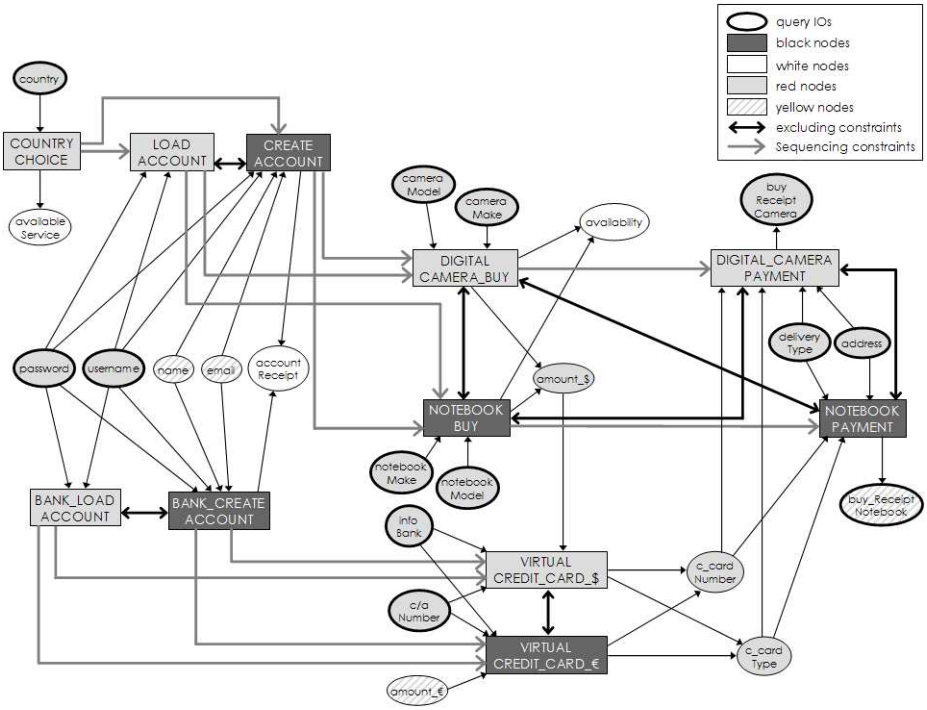

Fig. 3. Coloured graph for the first query.

Consider next the query specifying:

– inputs: username, password, camera_Model, camera_Make, notebook_Model, notebook_Make, c/a_Number, info_Bank, delivery_Type and address, and

– outputs: buy_Receipt_Camera and buy_Receipt_Notebook.

The algorithm proceeds similarly to the first considered case. The graph produced at the end of the matchmaking phase is the same as the one produced for the previous example and all query outputs are produced by at least an atomic process. During the second step, SAM paints in yellow all data nodes with the exception of available_Service, account_Receipt and availability as in the previous example. At the beginning of the "Red&Black Colouring" phase only the Bank_Load_Account process is FIRABLE and hence it is coloured in red and added to the PROCESS_SEQUENCE list, while the Bank_Create_Account process is painted in black. This step ends with a FAILURE as there are yellow query outputs but there are no FIRABLE processes. SAM continues then with the "Analysis of Triples" phase but is unable to find a set of FAILUREs that collectively are able to provide the buy_Receipt_Camera and buy_Receipt_Notebook outputs. Next, SAM queries the client whether it wants more information about the generated process sequence and about how it is possible to fully satisfy the query. Assuming that it agrees, SAM continues with step five ("Individuating Additional Inputs") when it looks for each unsatisfied output for yellow process nodes that generate this output. In our case, Digital_Camera_Payment and Notebook_Payment are the atomic process that can generate the buy_Receipt_Camera and buy_Receipt_Notebook outputs respectively. Yet, their execution is conditioned by the execution of their predecessors and moreover, both processes need c_Card_Number and c_Card_Type to be provided as inputs. A possible list of such additional inputs returned by SAM is: {country, c_Card_Number, c_Card_Type}. Indeed, all inputs needed for the execution of their predecessor processes are contained in the query with the exception of country.

It is worth noting that SAM is able to solve both queries presented in Section 2. SAM responds to the first query with the following PROCESS_SEQUENCE: [Country_Choice, Load_Account, Bank_Load_Account, Notebook_Buy, Virtual_Credit_Card_$, Notebook_Payment]. To the second query, SAM responds with: [[Country_Choice, Load_Account, Digital_Camera_Buy, Digital_Camera_Payment], [Country_Choice, Load_Account, Notebook_Buy, Notebook_Payment]].

4 Concluding Remarks

We have presented a new algorithm — called SAM (for Service Aggregation Matchmaking) — for the composition-oriented discovery of Web services. As already mentioned in Section 1, the main novel features of SAM are:

(1) to perform a fine-grained matching (at the level of atomic processes of services rather than at the level of entire services),
(2) to feature a flexible matching by returning partial matches and by suggesting additional inputs (when some query output cannot be produced by the services in the registry),

(3) to discover service compositions capable of satisfying a query, when no single service can satisfy it. In such cases SAM also explicitly returns the sequence of atomic process invocations that the client must perform in order to achieve the desired result.

The first semantics-based algorithm for Web service discovery using DAML-S ontologies was developed by Paolucci et al. [9]. Their algorithm performs a matching between service requests and service advertisements described as DAML-S service profiles. An assessment of the deployment of this algorithm to a UDDI registry was recently reported in [6], where WSDL service descriptions were enriched with semantics descriptions in the style of DAML-S service profiles. As we already noted in Section 2, the algorithm described in [9] is however limited to discovering a single service, and it does not address the issue of discovering service compositions. An algorithm for service discovery using service compositions was recently presented by Aversano et al. [1]. Their algorithm analyses DAML-S service profiles (as [9]) and it is also capable of performing a cross ontology matching (for service descriptions using different ontologies) as well as of matching service compositions (when no single service can fulfil the client request). Comparing SAM with [1], one may note that SAM analyses the process model of services to perform a finer-grained matchmaking, at the level of atomic processes inside services rather than at the level of entire services. Moreover, when no service composition can satisfy the query, SAM is also capable of suggesting additional inputs that would suffice to get a full match.

The first service discovery algorithm based on the analysis of DAML-S process models of services was proposed by Bansal and Vidal [3]. As we already discussed in Section 2, SAM extends [3] by considering both compositions and multiple executions of services. A preliminary version of SAM was presented in [5], where a first, limited form of service composition discovery was described. The current version of SAM described in this paper substantially extends [5] by introducing the dependency graph and its analysis, by providing a list of atomic process invocations, and by suggesting additional inputs when needed.

A semi-automatic approach to composite service discovery was recently presented by Liang et al. [7]. An interesting feature of it is the employment of constraint matching over a service dependency graph, where constraints may specify data dependencies as well as non-functional properties of services (such as Quality Of Service). Their approach is however based on UDDI registries, and hence the accuracy of the discovery is limited due to the absence of semantic information.

Our plans for future work include assessing SAM by experimenting it on large numbers of queries and service advertisements. While we have tested our Java implementation of SAM on several examples, an obstacle to running massive experiments is the lack of available OWL-S descriptions of services (only a few are publicly available in the W3C Web site). A promising approach to ease the generation of OWL-S descriptions of services may be to publicly deploy (to UDDI registries) supporting tools that facilitate such descriptions, as done for instance by Kawamura et al. [6] to promote the generation of DAML-S service profiles.

Another direction for future work is to extend the matching featured by SAM in order to deal with other attributes of services (including extra-functional ones) and the use of different ontologies. Our long-term goal is to develop a well-founded methodology to support the discovery, aggregation, and —when necessary— adaption [4] of services.

References

1. L. Aversano, G. Canfora, and A. Ciampi. An algorithm for web service discovery through their composition. In L. Zhang, editor, *IEEE International Conference on Web Services (ICWS'04)*, pages 332–341. IEEE Computer Society, 2004.
2. S. Bansal. *Matchmaking of Web Services Based on the DAML-S Service Model, Master Thesis*. University of South Carolina, 2002.
3. S. Bansal and J. Vidal. Matchmaking of Web Services Based on the DAML-S Service Model. In T. Sandholm and M. Yokoo, editors, *Second International Joint Conference on Autonomous Agents (AAMAS'03)*, pages 926–927. ACM Press, 2003.
4. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 3:45–54, 2005.
5. A. Brogi, S. Corfini, and R. Popescu. Flexible Matchmaking of Web Services Using DAML-S Ontologies. In *IBM Research Report. ICSOC 2004 Proceedings – Short Papers*, November 2004.
6. T. Kawamura, J. D. Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Public Deployment of Semantic Service Matchmaking with UDDI Business Registry. In S. McIlraith and D. Plexousakis, editors, *Third International Semantic Web Conference (ISWC'04), LNCS 3298*, pages 752–766. Springer-Verlag, 2004.
7. Q. Liang, L. N. Chakarapani, S. Y. W. Su, R. N. Chikkamagalur, and H. Lam. A Semi-Automatic Approach to Composite Web Services Discovery, Description and Invocation. *International Journal of Web Services Research*, 1(4):64–89, 2004.
8. OWL-S Coalition. OWL-S 1.1 release. `http://www.daml.org/services/owl-s/1.1/`.
9. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matchmaking of Web Services Capabilities. In I. Horrocks and J. Hendler, editors, *First International Semantic Web Conference on The Semantic Web, LNCS 2342*, pages 333–347. Springer-Verlag, 2002.
10. M. Papazoglou and D. G. (editors). Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.
11. UDDI. The UDDI Technical White Paper. 2000. http://www.uddi.org/.
12. W3C. Semantic Web Services Interest Group Charter. `http://w3c.org/2003/10/swsig-charter`.
13. W3C. Web Service Description Language (WSDL) 1.1. World Wide Web Consortium (2001), http://www.w3.org/TR/wsdl.
14. J. Yang. Web service componentization. *Communications of the ACM*, 46(10):35–40, 2003.