Title: A Graph-Based Approach for Semantic Process Model Discovery

Ahmed Gater, Daniela Grigori, Mokrane Bouzeghoub
Prism, Universite de Versailles Saint-Quentin en Yvelines
45 avenue des Etats-Unis, 78035 Versailles Cedex, FRANCE

ABSTRACT

One of the key tasks in the service oriented architecture that semantic web services aim to automate is the discovery of services that can fulfill the applications or user needs. OWL-S is one of the proposals for describing semantic metadata about web services, which is based on the OWL ontology language. Majority of current approaches for matching OWL-S processes take into account only the inputs/outputs service profile. This chapter argues that, in many situations the service matchmaking should take into account also the process model. We present matching techniques that operate on OWL-S process models and allow retrieving in a given repository, the processes most similar to the query. To do so, the chapter proposes to reduce the problem of process matching to a graph matching problem and to adapt existing algorithms for this purpose. It proposes a similarity measure used to rank the discovered services. This measure captures differences in process structure and semantic differences between input/outputs used in the processes.

Keywords: OWL-S processes, web services, semantic process models, process matching

INTRODUCTION

Semantic web services envision a greater access to services on the web and a more automatic support to their management by providing the ability to describe their semantics in a formal and machine-processable manner. One of the key tasks in the service oriented architecture that semantic web services aim to automate is the discovery of services that can fulfill user needs.

Majority of approaches for semantic web services discovery are based on formal descriptions of both service advertisements (published by providers in a registry) and queries, i.e., the service that a user is looking for.

These descriptions are written in the same description language (for example, OWL-S [Martin et al, 2004]) and defined in an underlying decidable description logic based ontology language, such as OWL-DL, and OWL-Lite. This way, description logic reasoning can be used to automatically determine services that semantically match a given service request based on the terminological concept subsumption relations computed in the underlying ontology. Examples of such logic-based approaches to semantic service matchmaking are the OWL-S UDDI matchmaker [Kawamura et al. 2003] and WSMO discovery approach [Keller et al. 2005].

OWL-S is one of the proposals for describing semantic metadata about web services that is based on the OWL ontology language. OWL-S covers three main parts: the service profile for advertising and discovering of services; the process model, which gives a detailed description of a service's actions and behavior; and the grounding, which provides details on how to cooperate with a service.
Current approaches for discovering semantic services take into account only the service profile, by exploiting the relations between inputs and outputs concepts in the ontology [Klusch et al. 2006]. As we will see in next section, discovering services using only their inputs/outputs is not sufficient for some applications. For instance, when searching services for integrating in a given application, the process model specifying how to interact with the provider service has to be taken into account in the discovery process. Other application example is the scientific workflow discovery, where scientists look in a workflow repository for existing workflows that could support their research. Recent works [Goderis et al. 2006, Goderis et al. 2009] showed that scientists require tools for service discovery based on their underlying process model. These applications show that the discovery based on the profile has to be complemented by integrating the process model. In our view, service discovery can be reformulated as a three-phase process (see Figure 1):

- Phase 1: profile-based selection which uses classical information retrieval techniques to select the first bunch of services based on keywords and vectorial model matching, possibly improved by some ontological knowledge;
- Phase 2: structure-based selection which exploits behavioral properties given in the business process specification, usually leading to graph or automata matching;
- Phase 3: service ranking of the results provided by the previous phase, based on constraints satisfaction and quality features.

This chapter is concerned by phase 2. We present matching techniques that operate on OWL-S process models and allow retrieving in a given repository, the processes most similar to the query. As process models are usually represented as graphs, the problem of process matching can be reduced to a graph matching problem. In this chapter we show how existing graph matching algorithms can be adapted and extended for this purpose. We propose a graph-based similarity measure that will be used to rank the discovered services. This measure captures differences in process structure and semantic differences between inputs/outputs used in the process. In this chapter we present our matching approach as a step in the process of service discovery, but it can be used in several applications (merging processes, delta analysis, auto completion mechanism for modeling processes, …) requiring the matching semantic process models.

In section 2, we present two motivating scenarios where process model matchmaking is necessary. Section 3 presents existing approaches for service retrieval and shows their drawbacks for the presented scenarios. In section 4, we give an overview on the OWL-S ontology, by focusing on the process models. In section 5, we show how the process model matchmaking is reduced to a graph matching problem. Section 6 shows how the graph matching algorithm can be used for OWL-S process model matchmaking. The matchmaking algorithm is illustrated through an example in Section 7. Finally, section 8 presents some open problems and conclusions.

**Motivating scenario**

With the increasing importance of processes in current information systems and service oriented architecture, we are witnessing the emergence of a growing need for tools allowing matching processes. Examples of such applications are numerous: web service integration, retrieving scientific workflow, delta analysis, version management, compatibility and replaceability, analysis of business protocols, behavior based service discovery...

Due to space limitations, we limit ourselves to the presentation of two scenarios: web service integration, retrieving scientific workflow.
**Web services integration.** Consider a company that uses a service to order office suppliers. Suppose that the company wants to replace its current partner or to find a new retailer (say WalMart or Target). Many existing services are programmed to exchange data according to a specified protocol. The allowed message exchange sequences are called process model and can be expressed for example using BPEL abstract processes [Andrews et al. 2007] or OWL-S (for process models consisting of semantically annotated web services). Thus the company will search for a service having a process model that is compatible with service’s process model. Among retailer services, the most compatible one has to be found. If the service is not fully compatible, the company will adapt it or will develop an adaptor in order to interact with the retrieved service. In both situations, the differences between the process models have to be automatically identified. In the former case, finding the most similar service allows to minimize the development cost. In the latter case, identifying automatically the differences between protocols is the first stage in the process of semi-automatically adapters development (see [Benatallah et al., 2005]). An approach for mediating two OWL-S process models (provider and requester) was proposed in [Vaculin et al. 2007]. But, complementary to this approach, a method is needed to retrieve within a repository of OWL-S services, those having the process models that are the most similar to the user query.

**Retrieving scientific workflows.** Collaboratories emerge on the web, allowing scientists to share and re-use the scientific workflows describing their experiments. As an example, myExperiment (www.myexperiment.org) is one of the current public workflow repositories containing hundreds of distinct scientific workflows, contributed by hundreds of users. While current retrieval facilities in workflow repositories are mostly limited to browsing and keywords based searches, recent works [Goderis et al. 2006, 7] elicited requirements for workflow discovery through an experiment with scientists and developers form the myGrid workflow environment. The experiments showed: (i) the willingness of users to share and reuse workflows, (ii) the limits of current retrieval facilities and (iii) that process based discovery is a required feature. As some of these repositories use ontologies to describe and discover services (see [Wroe et al. 2003] as an example in a bioinformatics domain), they require methods and tools to retrieve semantic based process models based on the underlying process description.

To achieve efficient process model matchmaking techniques, for these applications, we must tackle three issues:

- Identify similar activities in the two graphs by using all sources of evidence: their names, inputs and outputs.
- Detect granularity differences between the two graphs: the same functionality can be implemented by a developer as an activity, while another developer could decompose it in several activities. Figure 2 shows an example of such granularity differences. The query process specifies user update contact functionality as a single activity (UpdateContact), while the target one implements it as a sequence of two sub-activities (Authentication followed by AddContact).
- Define a similarity measure taking into account structural differences (sequencing order differences)

In the next section, we present existing approaches for service discovery and their weaknesses for the presented scenarios and the highlighted challenges.
Related works

In this section, we give an overview of existing approaches for service retrieval and show their limits for the presented scenarios.

Currently, Web services discovery in registers like UDDI or ebXML are based on a search by keywords or tables of correspondence of couples (key-value). Within the framework of the semantic Web, description logics were proposed for a richer and precise formal description of services. Since semantic models provide a shared agreement on the meaning of terms regardless the underlying format (syntax and structure), annotating requirements and capabilities of Web services with these models introduces an additional level of abstraction that can improve significantly the relevance of web service discovery engines. Specifically, these models allow the
definition of ontologies (for example OWL-S) which are used as a basis for semantic matching between a declarative description of the required service and descriptions of the offered services ([Paolucci et al. 2002, Bernstein et al. 2002, Benatallah et al. 2003]).

In [Paolucci et al. 2002, Benatallah et al. 2003], a published service is matched against a required service when the inputs and outputs of the required service match the inputs and outputs of the published service. Two inputs/outputs, which are concepts in a given domain ontology, match if they have the same type or one is a generalization of the other.

In [Kawamura et al. 2003], several filters are defined for service retrieval: the namespace, the domain of ontology that is used, types of inputs/outputs and constraints. The OWLS-MX matchmaker ([Klusch et al. 2006]) extends this approach by introducing approximate matching based on information retrieval similarity metrics applied to the features characterizing the concepts in the ontology.

The approaches presented in [Cardoso et al. 2003, Wu et al. 2005] provide a lightweight semantic comparison of profiles based on similarity assessment methods taking into account also operational properties like execution time, cost and reliability.

Recently, the World Wide Web Consortium (W3C) produced a standard set of Semantic Annotations for WSDL (SAWSDL) [Farrell et al. 2007], aiming at disambiguating the semantic of Web services description during discovery. For a survey on approaches for semantic matchmaking of web service interfaces, see [Bellur et al. 2008].

Authors in [Bansal et al. 2003] argue that the matchmaking based only on service profile (inputs and outputs of the service) is not sufficient as some outputs data may be produced only under certain internal conditions.

Service discovery based on keywords or on the service profile, even if enriched by semantic annotations, is still insufficient for many applications such as those presented in the previous section. Therefore, recent approaches [Nejati et al. 2007, Ehrig et al. 2007, Wombacher et al. 2004, Minor et al. 2007, Dijkman et al. 2009, Dongen et al. 2008] take into account the process model to enhance the service discovery engines accuracy.

In [Nejati et al. 2007], authors propose process model similarity metric for processes captured as hierarchical statecharts, where each state represents an atomic activity. The similarity between two activities is evaluated on the basis of their labels and on their behavioral semantics. The algorithm iteratively computes a similarity degree for every pair of activities by aggregating the similarity degrees between their neighbors (successor of a state). The algorithm iterates until either the similarity degrees between all state pairs stabilize, or a maximum number of iterations is reached.

In [Ehrig et al. 2007], authors propose a similarity measure for semantic process models specified using Petri nets. The proposed similarity is an aggregation of syntactic, linguistic and structural similarities of Petri net elements. Authors use Levenhstein distance [Gater et al. 2010] to evaluate the syntactic similarity of element label, while the linguistic similarity relies on WordNet dictionary to determine element synonyms. The structural similarity of two elements is defined as the weighted average of the similarities of all elements which may influence it.

In [Dijkman et al. 2009], authors reduce the process model (specified as rooted graphs) matchmaking problem into a graph matchmaking problem. They propose a structural similarity based on the notion of graph-edit distance [Messmer 1995]. The graph-edit distance is based on the minimum cost of edit operations (node/edge insertion and substitution) that are necessary to transform one graph to another. Moreover, authors define syntactic and lexical metrics to compute the similarity between the vocabularies of different models.
Another structure-based metric is proposed in [Minor et al. 2007] to evaluate the distance between two process models captured as rooted graphs. The distance is viewed as being the number of activities and control flow constraints that are not shared by the two process models. The distance between two process models $P_1$ and $P_2$ is expressed by the following formula: 

$$\delta(P_1, P_2) = |A_1| + |E_1| + |A_2| + |E_2|;$$

where $A_1$ (resp. $A_2$) are nodes within $P_1$ (resp. $P_2$) but not within $P_2$ (resp $P_1$), and $E_1$ (resp. $P_2$) are edges within $P_1$ (resp. $P_2$) but not within $P_2$ (resp. $P_1$).

In [Wombacher et al. 2004], authors give a formal semantics to the matchmaking of process model specified using finite state machines (FSM). In this approach, the process model matchmaking problem is formulated as FSM intersection, where two FSMs match if they share a non-empty set of common execution paths.

In [Dongen et al. 2008], it is proposed a technique for matching process models based on causal footprints. A causal footprint is defined as a tuple $P = (N, L_{lb}, L_{la})$, where $N$ is a finite set of activities, $L_{lb} \subseteq (P(N) \times N)$ is a set of look-back links, and $L_{la} \subseteq (P(N) \times N)$ is a set of look-ahead links. Each look-ahead link $(a, B) \in L_{la}$ means that the execution of $a$ is followed at least by the execution of an element $b \in B$. Each look-back link in $(A, b) \in L_{lb}$ means that the execution of $b$ is preceded at least by the execution of an element $a \in A$. Process models are represented as vectors, and their similarity is considered as the cosine of the angle between their respective vectors. The dimensions composing the vectors are activities, look-ahead links and look-back links belonging to at least one of the two causal footprints.

Other area of related works [Vaculn et al. 2007, Beeri et al. 2008, Awad et al. 2008] concerns process query languages allowing to retrieve in a repository process models satisfying structural constraints. In [Vaculn et al. 2007], a query language for services is defined, which allows to find services by specifying conditions on the activities composing them, the exceptions treated, the flow of the data between the activities.

BP-QL [Beeri et al. 2008] is a graph-based language for querying BPEL processes which allows to navigate along the path-dimension inside a single process graph and to zoom across process graphs of the used services.

BPMN-Q [Awad et al. 2008] is another query language for processes expressed using Business Process Modeling Notation (BPMN). In order to deal with heterogeneity of activity names, semantic expansion of BPMN-Q queries is realized by constructing new queries using similar activities to the ones employed in the query and returning results that match either the original query or any of its derivatives.

These query languages are based on an exact matching between the query pattern and the target process model. They not allow to retrieve patterns similar to the query, i.e. if a graph containing the query pattern does not exist, but a graph containing a similar one exist, it will not be returned to the user.

Complementary to the work presented here, are recent works that try to uncover the particular similarity metrics people use for establishing workflow similarity. Authors in [Bernstein et al.
2005] look for the best semantic similarity measures to rank business processes from the MIT Process Handbook, based on process ontology. The work presented in [Wombacher et al. 2006] seeks to elicit the similarity metrics used by workflow researchers when comparing the control flow of workflows described by Finite State Machines (FSMs). Both works show that a general similarity measure reflecting human similarity assessments can hardly be found, the first one suggesting personalizing the similarity measures according to the user’s similarity assessment style.

To summarize, the need to take into account the service process model in the retrieval process was underlined by several authors and some very recent proposals exist ([Nejati et al. 2007, Wombacher et al. 2004]).

However, most of these approaches do not consider all activity attributes that can make the calculation of distance between process activities more efficient. In this sense, we defined a similarity measure between process activities that takes into account all sources of evidence: inputs, outputs and name.

Moreover, granularity differences may exist between two processes: the same functionality can be implemented by a developer as an activity, while other developer could decompose it in several activities. While majority of existing approaches considers only simple matches (1-1 activity correspondences), we introduce in this chapter a technique that allows identifying complex matches (1-Many activity correspondences) between activities based on their inputs/outputs semantics and control flow dependencies.

An approximate matching technique addressing these issues is defined in [Grigori et al. 2008] for service retrieval based on BPEL process model. This technique is extended in [Gater et al. 2010] for semantic annotated process models (OWL-S process model). Hereafter, we present in details these techniques.

**Overview of OWL-S ontology**

OWL-S [Martin et al. 2004] is an ontology for web services description that enables greater access to web services by offering a set of markup language constructs for describing the properties and capabilities of Web services in an unambiguous, computer-interpretable form. OWL-S ontology covers three main parts: service profile, service grounding and, the service process model.

The profile gives mechanisms for specifying service functionality, provider information, and non-functional properties. Service functionality describes the function the service offers in terms of required inputs to produce the expected outputs. Furthermore, it describes the preconditions under which the service operates, and the effects resulting from the execution of the service. The provider information consists of the contact information about the entity offering the service; it may include geographic locations of the company offering the service, the operator responsible of marketing or running. The non-functional properties allow describing an unbounded list of service parameters that can contain service category, quality rating, geographic availability, time response, etc.

The grounding of a service specifies the details of how to access the service in terms of protocol and message formats, serialization, transport, and addressing. Concrete messages are specified explicitly in the grounding part. OWL-S adopts Web Services Description Language (WSDL) for concrete message specification. The process model gives a detailed description of a service’s
actions and behavior. It captures the temporal and logical dependencies (control flow) of process model activities. The elementary unit of the process model is an atomic process, which represents one indivisible operation that the client can perform by sending a particular message (possibly complex) to the service and receiving a corresponding response message (possibly complex). Atomic processes are specified by means of their inputs, outputs, preconditions, and effects (IOPEs). Types of inputs and outputs are defined as concepts in some domain ontology or as simple XSD data-types.

```
<process:CompositeProcess rdf:ID="BankCodes">
  <process:Choice>
    <process:sequence>
      <atomic process:GetBranchNumber>
      <atomic process: GetNationalCode>
    </process:sequence>
    <process:sequence>
      <atomic process:GetNationalCode>
      <atomic process: GetBankCounter>
    </process:sequence>
  </process:Choice>
</process:CompositeProcess>
....
  <process:hasInput>
    <process:input>&Bank;#BankName</process:input>
  </process:Input>
  <process:hasOutput>
    <process:Output>&Bank;#ABI</process:Output>
  </process:hasOutput>
</process:AtomicProcess>
<process:AtomicProcess rdf:ID="GetBranchNumber">
  <process:hasInput> ......
```

Figure 3. OWL-S process model specification example
Atomic processes can be combined into composite processes by using the following control constructs: Sequences, Any-Order, Choice, If-Then-Else, Split, Split-Join, Repeat-Until and Repeat-While.

**Sequence** describes a list processes to be executed in the order they appear. **Split** specifies a bag of processes to be executed concurrently. **Split** completes as soon as all of its sub processes have been scheduled for execution. **Split-Join** specifies the concurrent execution of its branches and completes when all of these branches have completed. **Any-Order** specifies a set of processes that have to be executed in some unspecified sequence but not concurrently. Processes of an **Any-Order** construct cannot overlap. **Choice** specifies a set of unconditional branches, where any of them may be chosen for execution. The **If-Then-Else** control construct describes a conditional branching. If the condition holds, the branch associated to Then is executed, otherwise, the Else branch is executed. A **Repeat-While** condition executes the process if the condition is true and loops, and exits if the condition is false. **Repeat-Until** executes the process, tests for the condition, exits if it is true, and otherwise loops.

Figure 3 sketches an example of OWL-S process model specification (BankCodes) taken from the bank domain. The BankCodes process model allows retrieving, through the bank name its local and international identifiers. The specification is divided in two parts. The first one specifies the process model (CompositeProcess XML element of the specification depicted in Figure 3), and the second one defines the atomic processes composing the composite one (AtomicProcess XML element of the specification shown in Figure 3). Figure 4 shows the ontologies used to annotate BankCodes atomic processes inputs and outputs.

**A Graph-based approach for process model similarity evaluation.**

In this section we show how the process model matchmaking is reduced to a graph matchmaking problem. By transforming query and target OWL-S process models to graphs, the process model matchmaking turns into a graph matching problem which is formulated as a search for graph or sub-graph isomorphism. To develop matching techniques allowing the delivery of inexact matches and the evaluation of the semantic distance between target models and user query, we adapt error-correcting sub-graph isomorphism (ECSI) detection algorithm [Messmer 1995] that integrates the concept of error correction into the matching process by using graph edit operations. The principle behind ECSI algorithm is to apply edit operations over the target graph until there exists a sub-graph isomorphism to the query graph. Each edit operation is assigned a
cost function depending on the application. The goal is then to find the sequence of edit operations leading to an isomorphism that minimizes the graph-edit distance.

**Background and basic definitions**

Our choice to represent the process models using graphs is motivated by the following reasons. First, the procedure to transform process models to graphs is simple (nodes represent service operations and edges represent constraints between them). Secondly, graphs are a general and powerful data structure for the representation of objects that have been successfully applied in different applications for evaluating objects similarities. In the following we give the definitions of error correcting graph matching as given by [29]. A directed graph is a set of labeled nodes that are related by directed edges. Its formal description is given by the Definition 1.

**Definition 1. Directed labeled graph.** Let L_V and L_E be the sets of node labels and edges labels respectively, a directed labeled graph is a tuple G = (V,E,α,β), in which:

- V is the set of vertices,
- E ⊆ V×V is the set of edges,
- α : V → L_V is the vertex labeling function
- β : E → L_E is the edge labeling function.

Given a graph G and a sequence of edit operations Δ = (δ_1, δ_2, ..., δ_k), the edited graph Δ(G) = δ_k(δ_{k-1}(...δ_2(δ_1(G))))... is obtained by applying successively the operations of Δ over the graph G.

Given a graph G, a graph edit operation δ on G is any of the following: substituting the label α(v) of vertex v by l, substituting the label β(e) of edge e by l’, deleting the vertex v from G, deleting the edge e from G, inserting an edge between two existing vertices, decomposing a vertex v into two vertices v_1, v_2 joining two vertices v_1, v_2 into a vertex v.

**Definition 2. Edited graph** Given a graph and a sequence of edit operations Δ = (δ_1, δ_2, ..., δ_k), the edited graph Δ(G) is Δ(G) = δ_k(δ_{k-1}(...δ_2(δ_1(G))))...).

**Definition 3. Ec-sub-graph isomorphism** Given two graphs G and G’, an error correcting (ec) sub-graph isomorphism f from G to G’ is a 2-tuple f = (Δ, f_Δ) where Δ is a sequence of edit operations and f_Δ is a sub-graph isomorphism from Δ(G) to G’.

For each edit operation δ, a certain cost is assigned C(δ). The cost of an ec-sub-graph isomorphism f = (Δ, f_Δ) is the cost of the edit operations Δ, i.e., C(Δ) = Σ_{i=1}^{k} C(δ_i).

The sub-graph edit distance is then defined to be the minimum cost taken over all sequences of edit operations that are necessary to obtain a sub-graph isomorphism.

**From OWL-S process model to a semantic process graph**

An OWL-S process model is transformed into an attributed graph, called semantic process graph. A semantic process graph, whose formal description is given by Definition 4, consists of a set of activity and connector nodes connected via edges labeled with optional guards. A semantic process graph has one START node marking the beginning of the process and can have multiple END nodes representing the termination of the process.
OWL-S atomic processes are represented by activity nodes that are described by their names, a set of inputs and a set of outputs. The formal description of an activity node is given by the Definition 5.

Connector nodes capture the execution logic of the process model and represent split and join rules of types XOR or AND. AND-Split connector triggers all of its outgoing concurrent branches which are synchronized by a corresponding AND-Join connector. The XOR-Split represents a choice between one of several alternative branches which are merged by the corresponding XOR-Join. Outgoing branches of a XOR-Split may be labeled by the branching conditions. A connector node has two attributes: connector type (CT) (AND-Split, AND-Join, XOR-Split, XOR-Join) and connector source (CS) that records the control structure captured by the node. Its formal description is given in Definition 6.

**Definition 4. Semantic Process Graph** is a tuple \( \text{SPG} = (A,C,E,P,O,AD,CD,\tau,\lambda,\varphi) \), where:

- \( A \) is the set of activity nodes,
- \( C \) is the set of connector nodes,
- \( E \subseteq ((A \cup C) \times (A \cup C)) \) is the set of edges,
- \( P \) is the set of branching conditions,
- \( O \) is the set of ontological concepts,
- \( AD \) is the set of activity descriptor,
- \( CD \) is the set of connector descriptor,
- \( \tau : A \rightarrow AD \) is a function that maps activity node to an activity description,
- \( \lambda : C \rightarrow CD \) is a function that maps a connector node to its type,
- \( \varphi : E \rightarrow P \cup \{\text{true}\} \) is a function that maps each edge to its branching condition or true (edge without condition).

**Definition 5. Activity Descriptor** is a tuple \( \text{AD} = (\text{Name},O_A,\text{In},\text{Out},\theta) \), where:

- Name is the name of activity,
- \( O_A \subseteq O \) is the set of ontological concepts,
- \( \text{In} \) is the set of input attributes,
- \( \text{Out} \) is the set of output attributes,
- \( \theta : (\text{In} \cup \text{Out}) \rightarrow O \) is the function that maps input/output attributes to ontological concepts.

**Definition 6. Connector Descriptor** is a tuple \( \text{CD} = (\text{Type},\text{ConnectorSource}) \), where:

- Type \( \in \{\text{AND-Split, AND-Join, XOR-Split, XOR-Join}\} \)
- ConnectorSource \( \in \{\text{If-Then-Else, Choice, Any-Order, Repeat, Split, Split-Join, Repeat-Until, Repeat-While}\} \)

To transform an OWL-S process model to a semantic process graph, we propose an algorithm similar to the flattering strategy algorithm presented in [Shapiro et al. 1981]. The general idea is to map structured activities to their respective process graph fragments. The algorithm traverses the nested structure of OWL-S control flow in a top-down manner and applies recursively a transformation routine specific to each type of structured activity. A sequence is transformed by connecting all nested sub-processes with edges; each sub-process is then transformed recursively to its process graph fragment. For a While control structure a loop is created between an XOR-Join and an XOR-Split, the edge is labeled with the loop condition. The graph representation of an If-Then-Else consists of a block of alternative branches, labeled with the branching conditions,
between an XOR-Split and an XOR-Join. The alternative branches of a Choice start from a XOR-Split and merge in a XOR-Join. A Flow is transformed into a block of parallel branches triggered by an AND-Split and synchronized with an AND-Join. An Any-Order structure is transformed to a block of parallel branches starting with an AND-Split and synchronized with an AND-Join. Figure 1 resumes the correspondences between OWL-S control flow structures and process graph fragments.

Figure 6 shows the transformation of the OWL-S process model specification given in Figure 3.

We observe that the same behavior can be specified, in OWL-S, using different syntactical structures. We defined five syntactic equivalence rules allowing transforming equivalent syntactical structures to the same graph fragments. These rules are the followings:

- Choice ( P₁ , Choice(P₂,P₃) ) ↔ Choice( Choice (P₁,P₂), P₃) ↔ Choice(P₁,P₂,P₃)
- Split-Join (P₁, Split-Join(P₂,P₃)) ↔ Split-Join(Split-Join(P₁,P₂), P₃) ↔ Split-Join(P₁,P₂,P₃)
- Split (P₁, Split (P₂,P₃)) ↔ Split ( Split(P₁, P₂,) P₃ ) ↔ Split (P₁,P₂,P₃)
- Any-Order(P₁,Any-Order(P₂,P₃)) ↔ Any-Order(Any-Order(P₁,P₂),P₃) ↔ Any-Order(P₁,P₂,P₃)
- If Cond₁ ( If Cond₂ (P₁) Else (P₂) ) Else P₁ , ↔ If ( (Cond₁ and Cond₂) (P₁) ) Else ((If (Cond₁ and !Cond₂) (P₂)) Else (If (!Cond₁) (P₃)) )

Figure 5 Correspondences between OWL-S control flow structures and graph fragments
OWL-S graph matchmaking

In this section we illustrate the use of the error-correcting graph matching algorithm for OWL-S process models matchmaking. We first give an overview of the matchmaking process and then we discuss each step in detail. Our OWL-S matchmaking approach, illustrated in Figure 7 is composed of the following steps. First, the OWL-S process models to be compared are transformed to graphs. Next, the similarity between the two graphs is evaluated by a similarity analyzer that is composed of the following modules (Figure 7):

- **Semantic comparison** module calculates the semantic similarity between the input/output concepts.
- **Linguistic comparison** module implements different algorithms to find the similarity between two strings, such as tokenization, NGram, Check synonym, Check abbreviation.
- **Ontological concepts similarity** module contains different metrics for measuring the degree of similarity of two concepts.
- **Comparison rules** module groups the cost functions for the graph edit operations.
- **Graph matchmaking** module takes as inputs the graphs produced by the OWL-S parser and finds out the error correcting sub-graph isomorphism with minimal cost.
- **Composition rules** module checks whether composition/decomposition operations are necessary and add their cost to the total distance.

In the next subsections, we present in details the functionalities of these modules.

**Semantic process graph matchmaking algorithm**

This module implements the algorithm for error-correcting sub-graph isomorphism detection ([Messmer 1995]). The sub-graph isomorphism detection is based on state-space searching using an algorithm similar to A* [Bunke 2000]. The basic idea of a state-space search is to have states representing partial solutions of the given problem and to define transitions from one state to another. Thus, the latter state represents a more complete solution than the previous state. For each state $s$ there is an evaluation function $f(s)$ which describes the quality of the represented solution. In the case of sub-graph isomorphism detection, given a target graph $G_T$ and a query graph $G_Q$, a state $s$ in the search space represents a partial mapping from $G_T$ to $G_Q$. Each partial mapping $P$ implies a number of edit operations ($\delta_1, \delta_i$) and their cost $C(P)$ is used to define the evaluation function $f(s)$.

In other words, the algorithm starts by mapping the first activity node of $G_T$ with all the activity nodes of $G_Q \cup \{\$\}$ (symbol $\$ is used to denote deleting activity nodes in the target graph $G_T$) with respect to the pruning function *feasible* (line 1 of Algorithm 1) and chooses the best mapping (with minimal cost) (Algorithm 1, line 6). This represents a partial mapping that will be extended by adding one activity node at a time (line 9). The process terminates when either a state representing an optimal ec-subgraph isomorphism from $G_T$ to $G_Q$ has been reached or all states in the search space have edit costs that exceed a given acceptance threshold.

The cost of the mapping $C(P')$ (line 9.2) represents the cost of extending the current mapping $P$ with the next activity node in $G_T$. Extending the mapping by matching an activity $b \in A_Q$ that is not yet mapped to an activity node $a$ in the target graph (that does not belong to the current
mapping) implies an activity node edit operation and an edge/connector edit operations: First, the attributes of $a$ must be substituted by attributes of $b$, and secondly, for each mapping $(a', b') \in P$ it must be ensured that any link (two activity nodes can be connected either by an edge or via a connector node) between $b'$ and $b$ in $G_Q$ can be mapped to a link between $a$ and $a'$ in $G_T$ by means of edge and connector edit operations. Since connector nodes express control flow constraints, they are compared in a manner similar to edges. Thus, when matching two graphs, we are not interested in finding all the correspondences between similar connector nodes, but comparing how mapped activity nodes are connected.

**Pruning the state-space search.**

The major problem with such an exhaustive algorithm is its complexity evaluated to $O(M^2N^2)$ in the best case (when the distance between the target and the query graphs is minimal) and $O(M^3N)$ in the worst case (when the two graphs are completely disparate). “$M$” and “$N$” are respectively the total number of nodes in the query and target graphs. Experiments shows that this algorithm can be used, with a low cost, for matching process graphs having less than 30 activities [Grigori et al. 2008]. Thus, to deal with larger process sizes, we should define effective criterion allowing reducing the search-space while guaranteeing finding satisfactory matchers.

![Figure 6 Transformation of the OWL-S process of Figure 2 to its SPG.](image)

Figure 6 Transformation of the OWL-S process of Figure 2 to its SPG.
The first criterion relates to the maximum cost that a mapping must never exceed. Thus, the mappings having a cost exceeding an acceptable threshold are dropped. When a complete mapping is found, its cost becomes the new acceptable threshold.

The second criterion concerns the maximum cost that a new pair of matches must never exceed. Clearly, it is not interesting to extend a mapping by adding a new pair of activity nodes that don't match at all. In the example shown by figure 8, matching the query activity "Login" against the target one "PayOff" is completely senseless.

Suppose that a state, representing a partial mapping $P$ between the query and the target graphs, is extended by matching a couple of activity nodes $(a,b)$ ($a \in A_T$ and $b \in A_Q$). This extension is not a promising one when the distance between $a$ and $b$ exceeds a given threshold $MatchThreshold(MDThreshold,SSThreshold)$. This threshold specifies the minimum similarity ($SSThreshold$) between two activity nodes and the minimal matching degree ($MDThreshold$) that inputs/outputs of $a$ and $b$ should respect to be added to $P$. However, deleting an activity is still permitted. Accordingly, the new matches that don't meet this pruning condition are discarded. feasible function (Algorithm 2) implements this pruning rule.

Our experiments [Gater et al; 2010] show that by setting the $MatchThreshold$ to $(PlugIn,0.5)$, and acceptable threshold to 10, our matchmaking algorithm can be used, with acceptable execution time, for process graphs having less than 40 activity nodes.

**Similarity metrics**
One of the major issues in our technique is the ability to determine the degree of resemblance of two activities. To estimate this similarity, we combine multiple sources of evidence: name, inputs and outputs. We make usage of linguistic similarity for computing activity names similarity, since they are human-assigned, and ontological similarity for evaluating the similarity of inputs/outputs similarity, since they are taken within a domain ontology.
Linguistic similarity. Activity names are human-assigned and are often formed by a combination of words and can contain abbreviations. Therefore, their similarity is computed using linguistic and textual similarity techniques. To obtain a linguistic distance between two names, we use existing algorithms: N-Gram, Check synonym, Check abbreviation and tokenization.

The tokenization algorithm tokenizes the name into its component words, removes stopwords, stems the words to their linguistic roots and then N-Gram, Check synonym and Check abbreviation algorithms are applied on each pair of tokens.

The N-Gram algorithm estimates the similarity according to a number of common q-grams between words [2]. The Check abbreviation algorithm uses a custom abbreviation dictionary to check whether a token is an abbreviation of another.

The Check synonym algorithm uses a linguistic dictionary to compute the similarity between tokens. We used in our implementation the free WordNet::Similarity API [Pedersen et al. 2004]. This API provides six similarity metrics implemented over the WordNet dictionary. These measures take as input two words, and return a value representing their similarity degree according to their relationships in the dictionary.
Algorithm 3. LinguisticSimilarity

1. INPUTS Token1, Token2
2. OUTPUTS Similarity
3. IF (Ngram(Token1, Token2)=1)
   Return 1;
4. ELSE IF (Synonyms(Token1, Token2)=1)
   Return 1;
5. ELSE IF (Abbreviation(Token1, Token2)=1)
   Return 1
ELSE
   Return {Ngram(Token1,Token2) + Synonyms(Token1,Token2) }/2

Algorithm 3 shows details of the routine that computes the similarity of two tokens. If two tokens are the same (Ngram function returns similarity value equal to 1), the LinguisticSimilarity routine returns 1. Otherwise, the LinguisticSimilarity routine checks the synonymy (Synonyms function of algorithm 3) of the tokens and return 1 if they are synonym. If the tokens are not synonym, the algorithm checks whether one token is an abbreviation of another and returns 1 if it’s the case. Finally, if the three functions values are between 0 and 1, the similarity of the tokens is the average of them.

There are other possible ways to measure name similarity: Levenshtein edit distance algorithm, techniques borrowed from the information retrieval area like TF-IDF or a combination of techniques can be considered. However, defining a clever function for syntactic similarity is outside the scope of this chapter, since the focus of our work is on behavior similarity.

Ontological similarity. Inputs and outputs are concepts taken from a domain ontology and therefore the similarity between two concepts c_q (input of the query) and c_t (input of the target) is defined on the basis of their semantic relationship in the ontology. We can distinguish three cases. The first case (Exact), which is trivial, c_q and c_t are the same concept and then their similarity is 1.
The second case is when $q$ is a specialization of $t$ (Plug-In); then the similarity is 1, since this guarantees that target activity will be executable with the provided input ($q$). The third case includes the cases when $q$ is a generalization of $t$ (Subsume) and the case when $q$ and $t$ have a common ancestor (Fail). In these cases, we use ontological concept similarity metric.

The similarity between two concepts is assessed using a pair of parameters, matching degree (MD) and semantic similarity (SS). The matching degree (MD) records the logical matching between two concepts. The MDs are ordered according to their levels of restriction: \textit{EXACT} > \textit{PLUG-IN} > \textit{SUBSUMES} > \textit{FAIL}. SS is a real number ranging between 0 and 1 and reflects the degree of resemblance between two concepts. Algorithm 4 describes the routine for computing concepts similarity.

Many metric similarities between ontological concepts have been proposed in past research. These measures can be divided into three main approaches [Song et al. 2007]:

- Feature-theoretical approach: the similarity between two concepts is related to both common and different characteristics. The more characteristic they share, the more they are similar.

- Edge-based approach: the similarity between two concepts is related to the path length between them. The shorter the path from one concept to another, the more similar they are. The links connecting two nodes are weighted to overcome the limitation imposed by the uniform distance.

- Information theoretic models: the similarity between two concepts is defined as the information content of the lowest common ancestors of them. The information content of a concept $c$ is defined as the negative log likelihood of the probability of encountering an instance of the concept. The intuition behind the use of negative likelihood is that the more probable a concept is of appearing, the less information it conveys.

Depending on the expressiveness of the used ontology, a metric can be preferred to another. These different metrics can be implemented in the Ontological concepts similarity metrics module. In our implementation, we use a well known path-based metric to compute the distance between two concepts $q$ (query concept) and $t$ (target concept):

$$
\text{OntologySim}(q,t) = 1 - 2 \ast \omega(\text{lca}(q,t)) - \omega(q) - \omega(t),
$$

where $\text{lca}(q,t)$ is the least common ancestor of $q$ and $t$, and $\omega(c)$ is the weight of the concept $c$ in the ontology.

Function \textit{ConceptSimilarity} is the basic block to calculate the similarity between two sets of concepts that is processed by \textit{MatchSet} function. \textit{MatchSet} calculates the best matching between two sets ($D_q, D_t$) of concepts that maximizes the total of semantic similarities (SS). The formula of \textit{MatchSet} is given bellow:

$$
\text{MatchSet}(D_q, D_t, SS) = \begin{cases} 
\text{Max}(\text{MatchSet}, SS(D_q - I, D_t - J) + \text{CS}(I, J), \text{SS}), & \text{if} D_q \neq \emptyset, D_t \neq \emptyset, I \in D_q, J \in D_t, \text{and} \emptyset \subseteq D_q \\
0, & \text{otherwise}
\end{cases}
$$

\text{MatchSet} function returns also the set \textit{MDSet} of mapped pairs $(I,J)$. This brings us to the definition of semantic comparison between two sets of concepts. The semantic similarity and the
Matching degree between two sets of concepts are, respectively, the average of semantic similarities and the minimum of matching degrees of all the concept-matched pairs within $MDSet$.

$$\text{ConceptSimilarity}(D_r, D_2).SS = \frac{\text{MatchSet}(D_r, D_2).SS}{\min(|D_r|, |D_2|)}$$

$$\text{ConceptSimilarity}(D_r, D_2).MD = \min_{(I_j, f)\in MDset} (\text{ConceptSimilarity}(I_j, f).MD)$$

**Comparison rules**

The Comparison rules module contains all the application-dependent functions allowing calculating the cost of graph edit operations. These functions are used by the graph matching module for calculating the distance between two graphs. In the following, we explain the cost functions used for OWL-S process model matchmaking. The cost for inserting, deleting edges and vertices can be set to a constant. The cost for editing vertices (activity and connector nodes) is presented below.

**Matching Activity Nodes.** The distance between two activity nodes is computed by the function *Activity Node Distance Evaluation* (see algorithm 5), and is computed on the basis of inputs, outputs and name similarities. Activity names similarities are computed by the module *Linguistic Comparison*, while inputs and outputs sets similarities are computed using *ConceptSetSimilarity* module. Finally, the distance between the two activities (*DistanceNode*) is the difference between 1 and the weighted average of inputs/outputs semantic similarity and names similarity. Weights $\omega_{io}$ and $\omega_{name}$ indicate the contribution of semantic similarity of Input/Output and name similarity respectively ($0 < \omega_{io} < 1$ and $0 < \omega_{name} < 1$). The function returns also the matching degree of the two Input/Output sets which is defined as the minimum of the two matching degrees of inputs and outputs.

**Matching connector nodes.** As mentioned above, the connector nodes define the execution logic of the process models. Thus, we are not interested in finding all the correspondences between similar connector nodes, but comparing how mapped activities are linked (by edge or via

<table>
<thead>
<tr>
<th>Algorithm 4. Function ConceptSimilarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INPUTS : $c_q, c_l$</td>
</tr>
<tr>
<td>2. OUTPUT: Sim : Struct(SS, MD)</td>
</tr>
<tr>
<td>3. IF $(c_q = c_l)$ THEN</td>
</tr>
<tr>
<td>Sim.SS = 1 ; Sim.MD = Exact</td>
</tr>
<tr>
<td>4. ELSE IF $c_q &gt; c_l$ THEN</td>
</tr>
<tr>
<td>Sim.SS = 1 ; Sim.MD = Plug-In</td>
</tr>
<tr>
<td>5. ELSE IF $c_q &gt; c_l$ THEN</td>
</tr>
<tr>
<td>Sim.SS = OntologySim($c_q, c_l$) ; Sim.MD = Subsumes</td>
</tr>
<tr>
<td>6. ELSE</td>
</tr>
<tr>
<td>Sim.SS = OntologySim($c_q, c_l$) ; Sim.MD = Fail</td>
</tr>
<tr>
<td>Return Sim</td>
</tr>
</tbody>
</table>
Algorithm 5. Function Activity Node Distance Evaluation

1. INPUTS: A (NameA, InputA, OutputA), B(NameB, InputB, OutputB)
2. OUTPUT: MatchingDegree ; ActivityDistance
3. LOCAL InSimilarity, OutSimilarity: Struct(MD, SS)
4. Calculate Name Similarity NameSimilarity = LinguisticSimilarity(NameA, NameB)
5. IF (NameSimilarity=0) return (Fail,1).
6. Calculate Input/Output Similarity
   5.1 InSimilarity = ConceptSetSimilarity(InputA, InputB)
   5.2 OutSimilarity = ConceptSetSimilarity(OutputB, OutputA)
6. IOSimilarity = (InSimilarity.SS + OutSimilarity.SS) / 2
6. DegreeMatch = min(InSimilarity.MD, OutSimilarity.MD)
7. Calculate Distance Node
   ActivityDistance = 1 – (ωio * IOSimilarity + ωname * NameSimilarity) / (ωio + ωname)
8. RETURN (DegreeMatch, ActivityDistance).

Algorithm 6 Function Composition

INPUTS: a, P, T’, OPEN
While T’ is not empty
{ FOR each couple of activities b₁ and b₂ ∈ T’
   { IF (Condition 3(b₁, b₂) is false)
      1.1 Remove b₁ and b₂ from T’.
      1.2 IF (Distance(a, b₁) < Threshold) Add P ∪ {(a, b₁)} to OPEN.
      1.3 IF (Distance(a, b₂) < Threshold) Add P ∪ {(a, b₂)} to OPEN.
   ELSE IF (Condition 1(b₁, b₂) and Condition 2(b₁, b₂))
      2.1 Remove b₁ and b₂ from T’.
      2.2 Add the composition result of b₁ and b₂ to T’.
   }
}

a connector node). Suppose that the current mapping p is extended by matching a against b (a ∈ AT and b ∈ G₀). Thus, for each mapping (a',b') ∈ P (a' ∈ AT and b' ∈ G₀) such that there is a link (link₁) between a and a' in graph G₁ and there is a link (link₂) between b and b' in graph G₀, it checks whether an edit operation is needed or not. link₁ can be either an edge or a connector node. Figure 8 shows an example of editing connector node. The current mapping {(Start, Start), (Login, SignIn)} is extended by matching FlightBooking (of the query) against FlightReservation (in the target). In the query graph, Login is linked to FlightBooking via an edge, while in the target graph SignIn is linked to FlightReservation via a connector node (of type Xor-Split). Thus, this mapping induces an editing link whose cost is the cost of deleting an edge and adding a connector. Figure 9 shows all cases that may be encountered when computing the cost of editing links (edge and connectors). Constants PA (parallel to any-order), BD (branch deletion), LI (loop insertion), AP (any-order to parallel), BI (branch insertion), CD (condition deletion), CI (condition insertion), LD (loop deletion), Cₑₐ (edge deletion), Cᵢ (connector
insertion), \( C_{cd} \) (connector deletion) \( C_{ei} \) (edge insertion). Since there is no consensus on the definition of similar behavior [Bernstein et al. 2005, Wombacher et al. 2006], this table allows the user to express his own perception of behavioral equivalences/differences by specifying its own edit operation costs.

**Granularity level comparison**

As shown by the example of Figure 2, the models to be compared may have different granularity levels for achieving the same functionality. The classical edit operations take into account only 1-1 matches between activity nodes of the two graphs. Using the example of the Figure 2, a matching algorithm based only on the classical edit operations would match the query activity UpdateContact with one of the two target activities Authentication and AddContact and delete the other.

To solve this kind of mismatches, we need edit operations that allow discovering both simple matches (1-1) and complex matches (1-many or many-1). A complex match specifies that a combination of activities (connected sub-graph) in one process model corresponds to a combination of activities in the other.

Given a process graph \( G \), these edit operations are the followings:

- Decomposing an activity node \( a \) into a set of activity nodes \( \{a_1 \ldots a_n\} \).
- Composing a set of activity nodes \( \{a_1 \ldots a_n\} \) into a single activity node \( a \).

Suppose that no satisfactory matcher is found for a target activity \( a \): \( \forall b_i \in T' \), \( \text{ActivityDistance}(a,b_i) > \text{threshold} \) (\( T' \) is the set of nodes that are not yet mapped in the current mapping), we then try to compose activities from \( T' \) to a find satisfactory matcher for \( a \).

The composition is done iteratively until the termination conditions of the composition are no longer met.

Thus, at each iteration, the activities of \( T' \) are composed 2-2 and only compositions respecting composition conditions (enumerated bellow) are treated and considered as potential matchers of \( a \). Each composition is then considered as an atomic activity and added to \( T' \). \( T' \) is ordered according to the distance between its elements and this activity \( (a) \). This procedure is reiterated until composition conditions are no more satisfied. At the end of this procedure, the activity node \( a \) (for which we look for satisfactory matcher) is then matched against the last composition result.

Given two nodes \( b_1(\text{Name}_1, \text{Input}_1, \text{Output}_1) \) and \( b_2(\text{Name}_2, \text{Input}_2, \text{Output}_2) \) that are in \( T' \), the composition is permitted when:

- **Condition 1**: \( b_1 \) and \( b_2 \) are composable iff they are in parallel (AND(\( b_1,b_2 \))), in conditional/unconditional (XOR(\( b_1,b_2 \))) or in sequence (SEQ(\( b_1,b_2 \))).
- **Condition 2**: \( b_1 \) and \( b_2 \) share at least one input or one output with \( a \):
  - \( (\text{Input}_1 \cap \text{Input}_a) \cup (\text{Output}_1 \cap \text{Output}_a) \neq \emptyset \)
  - \( (\text{Input}_2 \cap \text{Input}_a) \cup (\text{Output}_2 \cap \text{Output}_a) \neq \emptyset \)
- **Condition 3**: the cost of the composition of the \( (I + 1)^{th} \) iteration is less than the cost of the \( I^{th} \) one. The cost of the first iteration is the activity distance between activity node \( a \) and the first activity node in \( T' \). \( b_1 \) may be an activity node originally belonging to \( T' \) or the result of the previous compositions.

The composition of two activity nodes \( b_1(\text{Name}_1, \text{Input}_1, \text{Output}_1) \) and \( b_2(\text{Name}_2, \text{Input}_2, \text{Output}_2) \) leads to an activity node \( b_{12}(\text{Name}_{12}, \text{Input}_{12}, \text{Output}_{12}) \) defined as follows:
Name_{12} = \text{concat}(\text{Name}_1, \text{Name}_2) \\
\text{Input}_{12} = \text{Input}_1 \cup \text{Input}_2 - (\text{Output}_1 \cap \text{Input}_2) \\
\text{Output}_{12} = \text{Output}_1 \cup \text{Output}_2 - (\text{Output}_1 \cap \text{Input}_2)

The composition routine is given by the algorithm 6. When several synchronized activities of the query graph correspond to one activity in the target graph, a decomposition of this node is necessary. Therefore, the same approach can be followed to define the decomposition algorithm.

Matching example

The Figure 10 shows two process models (Query and Target) taken from the Bank domain. The Query process model is that of the figure 6. The example highlights the challenging aspects that are handled by our algorithm to provide an effective technique for process model similarity evaluation.

The first one is related to the mismatches between metadata (name, inputs and outputs) describing atomic processes. The rules we defined to compare atomic processes allow mapping atomic processes even if they are described differently. For instance, target atomic process \text{GetCINCode} is matched against query atomic process \text{GetNationalCode}. Whatever their names are not exactly the same and the output of the second process subsumes the output of the first, the two activities match approximately.

The second one is related to the granularity level differences between the two process models, where the atomic process \text{GetBankCounter} in the query is matched against the sequence of \text{GetCABCode} and \text{GetBBAN} activities in the target. The rules of compositions that are based on inputs/outputs semantic allow detecting with accuracy granularity differences.

The third aspect concerns behavior differences and equivalences. For instance, \text{GetBranchNumber} and \text{GetNationalCode} activities of the query are in sequence, while their corresponding activities in the target are in parallel. The algorithm detects and matches equivalent behaviors even if they are specified using different control constructs and it is able to consider user behavioral equivalences/differences perception (through the table of connector node edit operation costs).

Conclusion

In this chapter, we described a graph-based approach for semantic process retrieval. The approach was illustrated using OWL-S, but can be adapted to other semantic process models. Two motivating scenarios were presented showing the need to develop process matching techniques. In order to allow an approximate matching, a graph error correcting matching algorithm was adapted and extended. The matchmaking algorithm returns a similarity score and a matching degree (reflecting the terminological concept subsumption relationship between inputs/outputs of atomic processes); it outputs also the edit operations needed to transform one process into the other. These graph edit operations could be useful for applications that require the development of adapters to solve differences between process models describing business protocols of two web services to be integrated.
Figure 8 Example of edit connector

<table>
<thead>
<tr>
<th>Link(a,b)</th>
<th>And-Split/Join</th>
<th>XOR-Split/Join</th>
<th>XOR-Join/Join</th>
<th>Edge</th>
<th>No link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Split/Split-Join</td>
<td>Any</td>
<td>Split/Split-Join</td>
<td>Any</td>
<td>Link(a,b)</td>
</tr>
<tr>
<td>And-Split/Join</td>
<td>0</td>
<td>PA</td>
<td>BD</td>
<td>BD</td>
<td>Li</td>
</tr>
<tr>
<td>Any</td>
<td>AP</td>
<td>0</td>
<td>BI</td>
<td>BD</td>
<td>Li</td>
</tr>
<tr>
<td>XOR-Split/Join</td>
<td>If-then-else</td>
<td>BI</td>
<td>BI</td>
<td>0</td>
<td>CD</td>
</tr>
<tr>
<td>Choice</td>
<td>BI</td>
<td>BI</td>
<td>CI</td>
<td>0</td>
<td>Li</td>
</tr>
<tr>
<td>XOR-Join/Split</td>
<td>While/Repeat</td>
<td>LD</td>
<td>LD</td>
<td>LD</td>
<td>LD</td>
</tr>
<tr>
<td>Edge</td>
<td>C_{ed}+C_{si}</td>
<td>C_{ed}+C_{si}</td>
<td>C_{ed}+C_{si}</td>
<td>C_{ed}+C_{si}</td>
<td>0</td>
</tr>
<tr>
<td>No link</td>
<td>C_{ed}</td>
<td>C_{ed}</td>
<td>C_{ed}</td>
<td>C_{ed}</td>
<td>C_{ed}</td>
</tr>
</tbody>
</table>

Figure 9 node editing cost table
An interesting future extension consists in learning matchmaking parameters starting from user preferences and rankings in order to provide personalizable similarity measures.

While the approach presented in this chapter allows the matchmaking of two process models, optimization techniques (indexing, filtering) have to be proposed for the case when one process model has to be compared with a set of models. Recent developments in graph indexing could be adapted for this purpose.

![Matchmaking example diagram]

**Figure 10** Matchmaking example

**References**


KEY TERMS AND DEFINITIONS

Web service: is a decoupled software component allowing communication between heterogeneous applications.

Semantic web services: aim to provide mechanisms to represent the requirements and capabilities of Web services in an unambiguous and machine interpretable form.

Process model: captures the temporal and logical dependencies (control flow) of a bag of business activity.

Semantic process graph: is an abstract representation of OWL-S process models. It can be used to capture several process model descriptions.

Similarity metric: evaluates the degree of resemblance of two objects.

Process matching: provides a mapping between two process models and a similarity metric evaluating how they are similar.

Process discovery: designates techniques allowing retrieving in a given repository, the process most similar to the query.