Automated Discovery, Interaction and Composition of Semantic Web services

Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan

Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{katia,paolucci,anupriya,naveen}@cs.cmu.edu

Abstract. In this paper we introduce a vision for Semantic Web services which combines the growing Web services architecture and the Semantic Web and we will propose DAML-S as a prototypical example of an ontology for describing Semantic Web services. Furthermore, we show that DAML-S is not just an abstract description, but it can be efficiently implemented to support capability matching and to manage interaction between Web services. Specifically, we will describe the implementation of the DAML-S/UDDI Matchmaker that expands on UDDI by providing semantic capability matching, and we will present the DAML-S Virtual Machine that uses the DAML-S Process Model to manage the interaction with Web service. We will also show that the use of DAML-S does not produce a performance penalty during the normal operation of Web services.

1 Introduction

The numerous Web services in existence constitute a distributed computation framework in which information and services are provided on demand in a machine-processable manner[1]. Yet, given any arbitrary problem, it is unlikely that it will solvable by one of the available Web services; rather, the solution of the problem will probably require an agent[2] to integrate results provided by several services. The composition of Web services and the integration of the information provided by Web services is the holy grail of the Web services infrastructure. Emerging standards such as BPEL4WS and WSCI provide languages to specify how Web services work together to address a problem or information need. Similarly, from the realm of the Semantic Web, DAML-S specifies an ontology for the description of what a Web service does and how to interact with it,

[1] Our view of Web services is based on the definition given in [7]. *A Web service is a software system identified by a URI whose public interface and bindings are defined and described by XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by Internet protocols.*

[2] Throughout the paper, we refer to 'agent' as any software entity, possibly with problem-solving capabilities, including intelligent autonomous agents [15].

which in turn provides the basis for Web service composition. This constitutes the main thrust of our paper.

The problem of composing Web services can be reduced to three fundamental problems: the first one is to make a *plan* that describes how Web services interact and how the functionality they offer can be integrated to provide a solution of the problem. The second problem is the *discovery* of the Web services that perform the tasks required in the plan. The third problem is the management of the *interaction* with those Web services. Crucially, planning, discovery and interaction are strictly interconnected: a plan specifies the type of Web services to discover, but it also depends on the Web services that are available. Similarly, the interaction process depends on the specifics of the plan, but the plan itself depends on the requirements of the interaction.

These three subproblems also dictate a set of challenges that any Web service infrastructure supporting composition must address. To discover a Web service, the infrastructure should be able to represent the capabilities provided by a Web service and it must be able to recognize the similarity between the capabilities provided and the functionalities requested. The second challenge for Web service infrastructure is to support the interaction between Web services. In particular, it should enable the specification of the information a Web service requires and provides, the interaction protocol it expects and the low-level mechanisms required to invoke the Web service. While service discovery and invocation are assumed to be part of the infrastructure, we assume that planning is the responsibility of individual agents and not of the infrastructure[3].

The challenges mentioned above highlight the need for semantic as well as the syntactic interoperability provided by XML. Syntactic interoperability allows agents and Web services to identify only the structure of the messages exchanged, but it fails to provide an interpretation of the content of those messages.

The current Web services infrastructure focuses on syntactic interoperability. Two popular proposed standards are SOAP [31] and WSDL [10], which use XSD to represent message data structures. UDDI [30] is a repository of useful information about Web services, but it does not allow capability-based discovery of Web services. WSCI [4] and BPEL4WS [12] describe how multiple Web services could be composed together to provide a more complex Web service. However, their focus remains on composition at the syntactic level and therefore, does not allow for automatic composition of Web services.

Semantic interoperability is therefore crucial for Web services. It allows Web services to (a) represent and reason about the task that a Web service performs (e.g. book selling, or credit card verification) so as to enable automated Web service discovery based on the explicit advertisement and description of service functionality, (b) explicitly express and reason about business relations and rules, (c) represent and reason about message ordering, (d) understand the meaning of exchanged messages, (e) represent and reason about preconditions that are

[3] Technically, infrastructure components, such as brokers who do composition on behalf of other agents, can be captured in our vision as a special type of agent.

required to use the service and effects of having invoked the service, and (f) allow composition of Web services to achieve a more complex service.

Web services can draw naturally from research in the Semantic Web, which aims to express the content of Web pages and make it accessible to agents and other services. The Semantic Web provides a set of languages with well-defined semantics and a proof theory that allows agents to draw inferences over the statements in the language. The content of Web pages is expressed by referring to ontologies, which provide a conceptual model to interpret the content.

The Semantic Web has the potential to provide the Web services infrastructure with the semantic interoperability it needs. It can provide formal languages and ontologies to reason about service descriptions, message content, business rules and relations between these ontologies. In this way, the Semantic Web and Web services are synergistic: the Semantic Web transforms the Web into a repository of computer readable data, while Web services provide the tools for the automatic use of that data.

The vision that we pursue is the realization of *Semantic Web services*, which result from the integration of semantic metadata, ontologies, formal tools and the Web services infrastructure [19]. A Semantic Web service is a Web Service whose description is in a language that has well-defined semantics. Therefore, it is unambiguously computer interpretable, and facilitates maximal automation and dynamism in Web service discovery, selection, composition, negotiation, invocation, monitoring, management, recovery and compensation. Specifically, Semantic Web services rely on the Semantic Web to describe (1) the content of the messages that they exchange, (2) the order of the messages exchanged and (3) the state transitions that result from such exchanges. The result of using the Semantic Web is an unambiguous description of the interface of the Web service which is machine understandable and provides the basis for a seamless interoperation among different services.

The use of the Semantic Web to describe Web services has wide ranging consequences. It allows the description of additional properties of Web services, such as the quality of service and security constraints in a coherent and uniform way that is universally understood. Furthermore, and most importantly, the description of the states produced by the execution of the Web service is the basis for the description of its capabilities as a transformation from its inputs and preconditions, to its outputs and effects.

The contribution of this paper is to demonstrate how semantic information enables discovery and autonomous invocation of semantic Web services. Furthermore, we will show how service discovery and invocation support automatic composition of Web services. The rest of the paper is organized as follows. In Section 2 we will discuss the contribution of DAML-S to discovery and invocation of Semantic Web services. Furthermore, we will discuss different approaches to Web service discovery and a formal semantics for the DAML-S Process Model. In Section 3 we will concentrate on how DAML-S can be used for capability based discovery and we sketch how it can be used to improve on the UDDI registry. In Section 4 we will introduce the DAML-S Virtual Machine: a general purpose

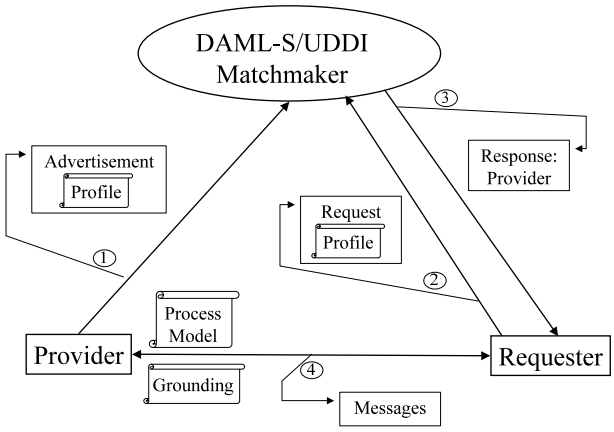

Fig. 1. Description of Web services Interaction

processor for the DAML-S Process Model that implements the formal semantics described in Section 2. In Section 5 we will provide a performance evaluation for the DAML-S Virtual Machine and we will show that DAML-S can be used with very little overhead. In Section 6 we describe an application that employ composition of DAML-S services and finally in Section 7 we conclude.

2 Semantic Web Services

The view adopted by DAML-S is that Web services have a set of capabilities that they offer to the rest of the community. The capability of some Web services may be to provide information, as for instance stock quotes, the capability of other Web services may be to provide services such as travel booking. In their normal operations, Web services, solve problems through information exchange or the exchange of services.

Each exchange minimally involves two parties: a requesting Web service that initiates the exchange and a provider that responds to the invocation. Since transactions may require more than one information exchange, the requester and the provider may dynamically switch their roles, for example the provider may ask the requester to select among alternative products in which case the provider is the one that seeks information from the requester. Indeed Web services cannot be restricted to a client/server relation but they should be able to interact as peers. Finally, some exchanges may involve more than two parties, for example, when the requester does not know which provider to invoke, the transaction involves one or more registries that receive a specification of the capability that

the requester expects in the provider, and select the provider whose capabilities match the needs of the requester.

A detailed view of the way DAML-S Web services interact is shown in Figure 1. The first task of a Web service is to advertise its capabilities with a registry (link 1), in our case the DAML-S/UDDI Matchmaker. The registration of capabilities allows the Web service to be discovered and to act as a provider. When a Web service needs to contact another Web service with a specific set of capabilities, it compiles the Profile of the ideal Web service it would like to contact, and submits it as a request to the Matchmaker (link 2). The task of the Matchmaker is to select the provider which declared a set of capabilities that more closely match the capability expected by the Requester. In our diagram, the Matchmaker located the Provider as the best match (link 3). Finally, the Requester knows about the Provider, and it can initiate the interaction (link 4). The interaction is regulated by the specifications in the Process Model and Grounding which define the interface of the Provider Web service.

In the rest of this section we provide a detailed description of DAML-S, and we will concentrate on two key problems for DAML-S and Semantic Web services in general: capability representation and the trade-offs that it entails, and the assignment of a formal semantics to the Process Model.

2.1 DAML-S

DAML-S Service Profile Service Profiles consist of three types of information: the capability of the service, a host of non-functional parameters, and a description of the person or legal entity that is responsible for the Web service. The capability of a Web service is represented as a transformation from the inputs and the preconditions of the Web service to the set of outputs produced (in return messages), and any (non message producing) side effects that result for the execution of the service. For example, a for-pay news reporting service might require as inputs a date and a credit card number; have as a precondition that the credit card number is a valid one and not overdrawn; have as output a Web page with news of that day, and have the effect that the specified credit card is charged. Functional attributes specify additional information about the service, such as the quality guarantees that it provides, or the cost of the service, or the classification of the service in some taxonomy such as the NAICS [8].

Implicitly, Service Profiles specify the intended purpose of the service, because they specify only the functionalities that the Web service is willing to provide publicly. For example, a book-selling service may involve two different functionalities: browsing to locate a book and selling the books found. The book-seller has the choice of advertising just the selling service or both the browsing and the selling functionality. The advertisements will affect its clients. If the book-seller advertises both browsing and selling, then clients interested only in browsing may contact the book-seller, while by advertising only the selling service, those same clients will not contact it. Ultimately, the decision as to which service to advertise determines how the service will be used.

DAML-S Process Model The DAML-S Process Model provides a more detailed view of the Web service than the Profile by showing a (partial) view of the workflow of the provider. The Process Model allows the requester to decide whether and how to interact with the provider. The requester may analyze the Process Model to verify whether the interaction with the provider leads to the results that were declared in the Profile. Through this analysis the requester detects the exceptions and possible failures that may emerge during the interaction and plan for contingencies. Finally, the Process Model reveals to the requester what information the provider requires and provides, and when to perform the information exchanges. Through the Process Model, the requester extracts the interaction protocol, and decides how to provide that information by using its own knowledge base, or by composing the invocation with other Web services.

In general, it is up to the provider to decide the degree of visibility that it allows of its own Process Model. The provider may decide that its own process is a "black box" in which case it will collapse all its processing in a single operation whose inputs the requester should provider and outputs are returned as answer. At the other extreme the provider may decide to provide a "glass box" view in which the requester has complete visibility on the workflow of the provider. In general, the provider should allow enough visibility to derive the interaction protocol. The result is a "gray box" where the requester has partial visibility on the process of the provider, but the provider hides some very important details.

The simplest units of description of the Process Model are the atomic processes which are equivalent to the basic functions performed by the provider. Atomic Processes can be composed into more complicated processes through workflow control structures such as sequence, if-then-else, or split and join. The Process Model provides a partially specified view of the provider because it allows the provider to hide details of its own workflow behind atomic processes. Furthermore, the Process Model provides non-deterministic constructs that can be specified only at execution time.

Atomic Processes also define the atomic units of interaction between the provider and the requester. The inputs of the atomic processes correspond to the information that the provider expects from the requester, and the outputs to the information that the provider sends to the requester. By following the control structures of the process model, the requester derives the sequences of information exchanges with the provider, which in turn correspond to the interaction protocol of the provider.

The Process Model and the Profile provide two different points of view of the same Web service. The Profile specifies the capabilities of the Web service (what the Web service does), while the Process Model provides a declarative specification of how the Web service achieves its goals, and how its requesters can interact with it. For example, the Profile says that a Web service, say Amazon's, sells books, the Process Model says that in order to buy books the requester needs to find the book, provide payment information, provide shipping address and so on before the book is actually delivered.

The two representations are used at two different times during the composition process: the Profile is used during discovery, when the requester knows what it expects from a provider, but it does not know what providers are available nor what processes do they perform. Upon matching, the requester can use the Process Model to select the most appropriate provider and to interact with and derive the provider's interaction protocol.

DAML-S Grounding The role of the DAML-S Grounding is to separate the abstract information exchange described by the Process Model from the implementation details, message format and so on. The DAML-S Grounding is responsible for mapping atomic processes into WSDL operations in such a way that the execution of one atomic process corresponds to the invocation of an operation on the server side. In addition the Grounding provides a way to translate the messages exchanged into DAML classes and instances that can be referred by the Process Model.

2.2 Capability Representation

Capability representation emerges as a key problem for Semantic Web services because any service requester may be aware of services it needs, without knowing precisely whether they are available on the Web or how to locate them. For example, a Web service that provides financial advice may need the latest quote of the IBM stock. To this extent, the Web service should transform the particular problem, i.e. get the quotes of the IBM stock, to a description of the capabilities it expects from the stock quotes provider, i.e. stock market reporting. Finally, it should use that capability description to locate the stock reporting Web service using a registry that can perform capability matching such as the DAML-S/UDDI Matchmaker.

A number of capability representation schemes have been proposed by the Semantic Web services community. Specifically, we distinguish between two types of representation schemes: the first one assumes ontologies that provide an *explicit* representation of the tasks performed by Web services. In those ontologies, each task is described by a different concept, while Web services capabilities are described by enumerating the tasks that they perform. The second representation scheme describes Web services by the state transformation and the information transfer that they produce. In this case, there is no mention of the task performed by the Web service; the task is *implicitly* represented by the state transformation and the Web service's inputs and outputs.

The two approaches to capability representation provide two ways to use ontologies. The schemes that make an explicit use of tasks require ontologies that assign a concept for each task performed by Web services, but since Web services can perform many different tasks, these ontologies can grow very large thus becoming unmanageable and may not scale up when new Web services with new capabilities become available. The implicit representation schemes do not suffer from those shortcomings since they require only concepts that describe the

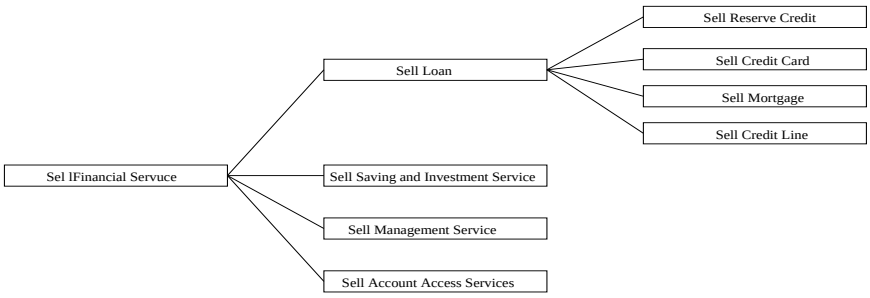

Fig. 2. Fragment of ontology of loan selling tasks

domain of the Web service, and then describe the Web service by the transformation it produces on its environment. On the other hand, explicit representations facilitate the matching process since there is no need to infer the task from its implicit representation. Each capability representation scheme strikes a different balance between the two extremes depending on the ontologies that it has available, and how closely they describe the capabilities of Web services.

Explicit Capability Representations An example of ontology which provides an explicit description of tasks and processes is the MIT Process Handbook [17]. Figure 2 shows a fragment of the specialization hierarchy of the ontology of tasks with the root SELL FINANCIAL SERVICES [17]. Furthermore, it shows that the concept `Sell Loan` is a specialization of SELL FINANCIAL SERVICE which in turn is specialized by SELL CREDIT CARD, `Sell Mortgage` and other concepts. In turn, the ontology associates to each process properties such as PORT that describes the I/O behavior of the process, and DECOMPOSITION that describes how the process is realized by the composition of other processes described in the ontology. The MIT Process Handbook can be used to index Web services for later retrieval [6]. For example, a Web service that sells loans would be associated with the concept SELL LOAN in the taxonomy in Figure 2.

The advantage of the explicit approach is that it reduces the burden of modeling Web services capabilities, since they can be represented by the list of tasks that they perform. The disadvantage of this representation, at least in principle, is that it is impossible to distinguish between Web services that sell loans whose amount is greater that $50,000 from those that sell loans whose amount is smaller than $10,000. To represent these constraints on the loan amount that the two Web services offer would require, at least in principle, the definition of two sub-classes of `Sell Loan` to describe the two different cases.

Implicit Capability Representation DAML-S Service Profile, as described above, adopts the implicit representation of capabilities of Web services. Web services are represented in terms of the transformation that they produce. Specifically, Web services are represented by the information transformation that they

produce in terms of inputs required and outputs generated, and by the state transformation produced in terms of preconditions that need to be satisfied for an execution and effects generated. In addition it provides a host of features that help with the specification of non-functional parameters of the Web service such as quality guarantees. An example of a capability specification for a stock reporting Web service is given in Figure 3. The input of the Web service is the ticker symbol, and its output is a quote for that ticker. The precondition is a valid account to which charge the giving of the information, and the effect is that the account is charged[4].

```
<profile:input>
  <profile:ParameterDescription  rdf:ID="Ticker_input">
    <profile:restrictedTo rdf:resource="Financial:Ticker"/>
  </profile:ParameterDescription>
</profile:input>

<profile:output>
  <profile:ParameterDescription  rdf:ID="Quote_Output">
    <profile:restrictedTo rdf:resource="Financial#Quote"/>
  </profile:ParameterDescription>
</profile:output>

<profile:precondition>
  <profile:ParameterDescription rdf:ID="valid_membership">
    <profile:restrictedTo rdf:resource="Financial#valid(account)"/>
  </profile:ParameterDescription>
</profile:effect>

<profile:effect>
  <profile:ParameterDescription rdf:ID="charged_account">
    <profile:restrictedTo rdf:resource="Financial#charged(account)"/>
  </profile:ParameterDescription>
</profile:effect>
```

Fig. 3. Examples of Input, Output, Precondition and Effect in DAML-S

The advantage of the implicit representations scheme is that any capability can be represented with no requirement of ontologies that explicitly classify capabilities. Furthermore, constraints on the capability of the Web service can easily be expressed. For example, the two sell loans services mentioned above can be represented by adding a precondition that the loan is smaller than $10,000 or bigger than $50,000. The problem of this representation is that the more

[4] At the time of writing, DAML does not support a rule language, therefore, conditions like VALID(ACCOUNT) or CHARGED(ACCOUNT) cannot be expressed.

constraints that are expressed about a Web service and its capabilities the more difficult is to perform a match of capability with the request.

Ultimately, when explicit ontologies are available, their use is bound to provide a more precise and efficient capability representation and matching. But, explicit representations can be used effectively only in limited domains, and they cannot scale up to the whole Web. We believe that implicit representations are the only way to represent Web services capabilities which has the potential to generalize to all services on the Web.

Combining Implicit and Explicit Representations The DAML-S Profile is a DAML class; as such it can be subclass ed and it can become part of a taxonomy of concepts. Therefore, it is possible to construct a taxonomy of profiles where each profile corresponds to a type of capability. Such a taxonomy would be equivalent to the taxonomy of services in the MIT Handbook. Indeed, it would be possible to construct a taxonomy equivalent to the taxonomy shown in Figure 2 where SELLFINANCIALSERVICES, is a subclass of Profile that specializes in the representation of a type of financial services. This class could also be sub-classed into SELLLOAN and so on.

The result is an hybrid representation of capabilities in DAML-S, where the representation of capabilities on the basis of input, outputs, preconditions and effects, can be augmented with the use of explicit ontologies of tasks. The advantage of this representation is that it maintains the expressive power of the implicit representation while facilitating the capability matching.

2.3 Execution Semantics for Service Composition

DAML-S provides only the specification of Semantic Web services. This specification must be complemented by (a) an execution model that preserves the DAML-S semantics, (b) an implemented computational architecture that enables dynamic, run-time semantic service discovery, interaction, interoperation and composition across the Web.

In this paper we adopt the operational semantics proposed in [1] which precises the execution behavior of a set of core constructs in the Process Model such as sequences, if-then-else conditionals and spawning of concurrent processes. From the semantics of these core constructs, the semantics of composite constructs such as loops, can be easily derived. In Section 4 we will describe in detail the DAML-S Virtual Machine, a computational architecture which implements these operational semantics.

An alternative semantics for the DAML-S Process Model has been proposed by Narayanan et al. [22], which describes the semantics of processes and their inputs, outputs, preconditions and effects as axioms in situation calculus. These axioms are mapped onto Petri net representations, which then describe the execution semantics of the DAML-S control constructs. The operational semantics we adopt uses a single representational model, namely that of functions, and is better suited for our purposes. The two semantics are equivalent in most respects, except for a few minor differences noted in [1].

$$(\text{FUNC})\ \frac{\phi \in \Omega}{\Pi, (E[\phi v_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[\phi_{\mathcal{A}} v_1 \cdots v_n], \varphi)}$$

$$(\text{APPL})\ \frac{\text{free}(u) \cap \text{bound}(e) = \emptyset}{\Pi, (E[(\backslash x \ \text{->}\ e)\ \ u)], \varphi) \longrightarrow \Pi, (E[e[x/u]], \varphi)}$$

$$(\text{CONV})\ \frac{y \text{ is a fresh free variable}}{\Pi, (E[\backslash x \ \text{->}\ e], \varphi) \longrightarrow \Pi, (E[\backslash y \ \text{->}\ e[x/y]], \varphi)}$$

$$(\text{SERV})\ \frac{s x_1 \cdots x_n := e \in \mathcal{S}}{\Pi, (E[s v_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[e'[x_1/v_1, \ldots, x_n/v_n]], \varphi)}$$

Table 1. Semantics of DAML-S Core - I

The formal semantics of the DAML-S Core is shown in Tables 1 and 2 where an inference rule of the form: A/B represents the drawing of the conclusion B on the basis of the premise A. The $\longrightarrow$ denotes a state transition, formally, $\longrightarrow \subset \mathit{State} \times \mathit{State}$ and we will write $s \longrightarrow s'$ to denote the transition from state s into state s'. The expression $\Pi, (E[\phi], \varphi)$ indicates that there is a set of processes Π that may be running concurrently, and $(E[\phi], \varphi)$ identifies the execution of one of such processes, where an operation of type ϕ is evaluated with a set of ports φ. A rule $A/(E[\phi], \varphi) \longrightarrow S$ specifies that if A is true, the execution of ϕ leads to state S[5].

The meaning of the rules in the two tables is the following. The rule FUNC in Table 1 specifies the effect of executing an atomic process. More precisely, it claims that if a process ϕ belongs to the set Ω of atomic processes, its execution results in the invocation of a corresponding operation $\phi_{\mathcal{A}}$ on the provider Web service. The rule SEQ in Table 2 specifies that the execution of a process e should wait on the return of value v from the previous process. Effectively, the rule forces the processes to be executed in a strict sequence. The evaluation of SPAWN e results in a new parallel process being created and in the return of the current process. The rule COND-TRUE specifies that if the condition evaluates to TRUE, then the first process is executed. This rule has a symmetrical COND-FALSE that specifies what happens when the condition is false. Finally CHOICE-LEFT specifies that if the execution of a process e_1 produces a change of state, then the same state is reached by executing [*choice* e_1 e_2] and choosing (non-deterministically e_1). The rule CHOICE-RIGHT is symmetrical. The other rules fill in technical details, specifically, PORT, REC, SEND deal with the introduction of new ports, and with sending and receiving messages.

[5] For a more detailed explanation of the formalism we refer the reader to [1].

$$\text{(SEQ)}\ \frac{-}{\Pi, (E[\texttt{return}\ v\ \texttt{>>=}\ e], \varphi) \longrightarrow \Pi, (E[(e\ v)], \varphi)}$$

$$\text{(SPAWN)}\ \frac{-}{\Pi, (E[\texttt{spawn}\ e], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ ()], \varphi), (e, \emptyset)}$$

$$\text{(PORT)}\ \frac{p \text{ new } \texttt{PortRef} \qquad \varphi'(x) = \begin{cases} \epsilon & \text{if } x = p; \\ \varphi(x) & \text{otherwise.} \end{cases}}{\Pi, (E[\texttt{newPort}\ \tau], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ p], \varphi')}$$

$$\text{(REC)}\ \frac{p \in Dom(\varphi) \qquad \varphi(p) = v \cdot w \qquad \varphi'(x) = \begin{cases} w & \text{if } x = p; \\ \varphi(x) & \text{otherwise.} \end{cases}}{\Pi, (E[p?], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ v], \varphi')}$$

$$\text{(SEND)}\ \frac{p \in Dom(\varphi_2) \qquad \varphi_2(p) = w \qquad \varphi_2'(x) = \begin{cases} w \cdot v & \text{if } x = p; \\ \varphi_2(x) & \text{otherwise.} \end{cases}}{\Pi, (E[p!\mathrm{v}], \varphi_1), (e, \varphi_2) \longrightarrow \Pi, (E[\texttt{return}\ ()], \varphi_1), (e, \varphi_2')}$$

$$\text{(COND-TRUE)}\ \frac{-}{\Pi, (E[\texttt{cond True}\ e_1\ e_2], \varphi) \longrightarrow \Pi, (E[e_1], \varphi)}$$

$$\text{(CHOICE-LEFT)}\ \frac{\Pi, (E[e_1], \varphi) \longrightarrow \Pi', (E[e_1'], \varphi')}{\Pi, (E[\texttt{choice}\ e_1\ e_2], \varphi) \longrightarrow \Pi', (E[e_1'], \varphi')}$$

Table 2. Semantics of DAML-S Core - II

The rules APPL, CONV, SERV deal with the management of variables and constants.

3 Matching Engine

In the previous sections we concentrated on the theoretical framework underlying the use of DAML-S for Web service discovery, interaction and composition. We described the structure of DAML-S, its approach to capability representation and we provided a formal semantics of the Process Model. On the basis of this theoretical framework, we implemented a computational model for the processing of DAML-S descriptions. Specifically, in this section, we will describe the Matching Engine that is at the core of the DAML-S/UDDI Matchmaker. The Matching Engine uses the capability description provided by DAML-S, to enhance UDDI with capability matching. In the next section we will describe

the DAML-S Virtual Machine: a general processor that allows Web services to interact using only DAML-S descriptions of Web services.

The task of the Matching Engine is to select the advertisements that match a given request. An advertisement matches a request, when the capabilities described by the advertisement are *sufficiently similar* to the capabilities requested. Of course, the problem of this definition is to specify what "sufficiently similar" means. In its strongest interpretation, an advertisement and a request are "sufficiently similar" when they describe exactly the same service. This definition is too restrictive because it is unlikely that there exists a service that satisfies all the needs of the requester.

To accommodate a softer definition of "sufficiently similar" we need to allow the matching engines to perform *flexible* matches, i.e. matches that recognize the degree of similarity between advertisements and requests. Service requesters should also be allowed to decide the degree of flexibility that they grant to the system. If they concede little flexibility, they reduce the likelihood of finding services that match their requirements, i.e. they minimize the false positives, while increasing the false negatives. On the other hand, by increasing the flexibility of match, they achieve the opposite effect: they reduce the false negatives at the expense of an increase of false positives.

An additional problem related with performing flexible matches is that the Matching Engine is open to exploitation from providers and requests. Service providers could provide capability advertisements that are too generic in an attempt to maximize the likelihood of matching. For instance, a service may advertise itself as a provider of everything, rather than be honest and precise regarding what service it provides. The matching engine must protect against such attempted exploitations by ranking advertisements on the basis of the degree of match with the request.

In a nutshell, the matching engine must satisfy the following desiderata:

- The matching engine should support flexible semantic matching between advertisements and requests on the basis of the ontologies available to the services and the matching engine.
- Despite the flexibility of match, the matching engine should minimize false positives and false negatives. Furthermore, the requesting service should have some control on the amount of matching flexibility it allows to the system.
- The matching engine should encourage advertisers and requesters to be honest with their descriptions.
- The matching process should be efficient: it should not burden the requester with excessive delays that would prevent its effectiveness..

The algorithm we propose strives to satisfy all four desiderata. Semantic matching is based on DAML ontologies: advertisements and requests refer to DAML concepts and the associated semantics. By using DAML, the matching process can perform inferences on the subsumption hierarchy leading to the recognition of semantic matches despite their syntactic differences and difference in modeling abstractions between advertisements and requests.

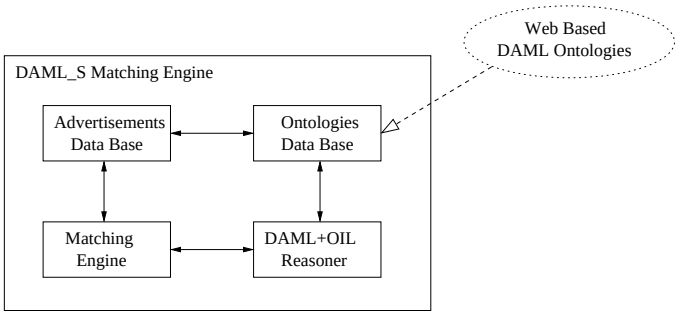

Fig. 4. The architecture of the DAML-S Matching Engine

The use of DAML also supports accuracy: no matching is recognized when the relation between the advertisement and the request does not derive from the DAML ontologies used by the registry. Furthermore, the semantics of DAML-S descriptions allows us to define a ranking function which distinguishes multiple degrees of matching.

Finally, the matching process is necessarily a complex mechanism that may lead to costly computations. In order to increase efficiency, the algorithm described here adopts a set of strategies that rapidly prune advertisements that are guaranteed not to match the request, thus improving the efficiency of the overall matching engine while maintaining its precision.

3.1 Algorithm for capability matching

The matching process for DAML-S [26] recognizes a match between the advertisement and the request, when the advertised service could be used in place of the requested service. Operationally this is correct when the outputs of the advertisement are equivalent or more general of the outputs of the request (ie, the advertised service provides all the information that the request needs), and when the inputs of the request are equivalent or more general of the inputs of the request. More formally, if in_{Ad} and in_{Req} represent the inputs of the advertisement and the request respectively, and out_{Ad} and out_{Req} represent their outputs then the matchmaker recognizes an *exact output match* when $out_{Ad} = out_{Req}$ and an *exact input match* when $in_{Ad} = in_{Req}$ Also, the matchmaker recognizes a *plugIn match* when $out_{Ad} \sqsupset out_{Req}$ or $in_{Req} \sqsupset in_{Ad}$.

When the outputs of the advertisement are more specific than the outputs of the request, then the advertised service provides less information than the requester needs. Still, it may be that the information provided by the advertiser is all that the requester needs, or that the requester may find another provider for the remaining data. In these cases, the matchmaker recognizes a *subsumed* match. Formally, the matchmaker recognizes a *subsumed match* when $out_{Req} \sqsupset out_{Ad}$ or $in_{Ad} \sqsupset in_{Req}$. When neither of the conditions above succeeds, there is no relation between the advertisement and the request and the match fails.

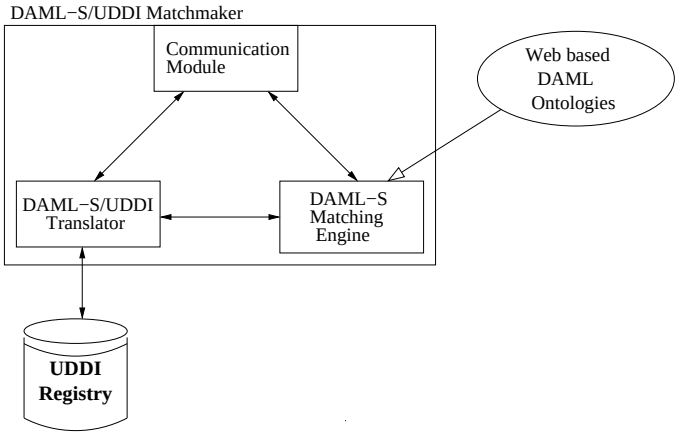

Fig. 5. The architecture of the DAML-S/UDDI Matchmaker

As should be clear from the discussion above, the matching algorithm defines a scoring function based on the degree of match detected. The scoring function is ordered in the following way: $exact > plugIn > subsumed > fail$. Furthermore, in general the Matchmaker prefers output matches over input matches. This is because the requester knows what it expects from the provider, but it cannot know what the provider needs until the provider is actually selected. Input matching is therefore relegated to a secondary role of tie breaker among providers with equivalent outputs.

Implementation of the Matchmaker The matching algorithm has been implemented in the DAML-S Matching Engine whose architecture is displayed in Figure 4. Advertisements are stored in the AdvertisementDB component and indexed using ontologies downloaded from the Web and stored in the OntologyDB.

Upon receiving a request, the Matching Engine component selects advertisements that are relevant for the current request; then it uses the DAML+OIL Reasoner and the ontologies downloaded from the Web to select the advertisements that really match the request and compute the degree of such match.

3.2 Adding Capability Matching to UDDI

The Universal Description Discovery and Integration (or UDDI) [30] is an Internet wide registry of Web services. Because of its strong industry backing, UDDI is expected to become the standard registry for Web services. UDDI allows businesses to register their contact points, and a host of useful information about Web Services such as binding information to allow Web services to interact.

In addition, UDDI supports the association of an unbounded set of properties to the description of Web services via a construct called TModel. For example, a

service may specify its category using the North American Industry Classification System (NAICS). While TModels support the association of any type of data with the advertisement, and their meaning is not codified, therefore two different TModels may have the same meaning, but this similarity cannot be recognized.

UDDI supports only a keyword based search of businesses, services and TModels in its repository. In addition services can be searched by their type specification through TModels. For instance, it is possible to search for all the services that adhere to the WSDL representation or that have a specific value associated with a TModel. Since search in UDDI is restricted to keyword matching, no form of inference or flexible match between keywords can be performed.

The limitation of UDDI is that it lacks an explicit representation of the capabilities of the Web service. The result is that UDDI supports the location of essential information about the Web service, *once it is known that the Web service exists*, but it is impossible to locate a Web service only on the basis of what it does. To solve this problem, a translation function from DAML-S Profiles to UDDI record has been implemented [25]. At its core, this function defines a set of TModels that correspond to properties of DAML-S Profiles therefore allowing any DAML-S profile to be recorded as a UDDI record.

The DAML-S/UDDI Matchmaker uses the translation function described above to map DAML-S advertisements into UDDI records, and then it uses the UDDI registry to store and retrieve them. Furthermore, leveraging on DAML-S capability representation, the DAML-S/UDDI Matchmaker adds a semantic layer that performs a capability matching between advertisements and requests using the matching engine described above and DAML ontologies published on the Web. The result of this work is empowering UDDI with DAML-S capability representation and with the corresponding matching mechanism to select Web services on the basis of their capabilities.

The architecture of the combined DAML-S/UDDI Matchmaker is described in Figure 5. The Matchmaker receives advertisements and requests through the *Communication Module*; upon recognizing that a message is an advertisement, the Communication Module sends it to the *DAML-S/UDDI Translator* that constructs a UDDI service description using information about the service provider, and the service name. The result of the registration with UDDI is a reference ID of the service. This ID combined with the capability description of the advertisement are sent to the DAML-S Matching Engine that stores the advertisement for capability matching. Requests follow the opposite direction: the Communicator Module sends them to the DAML-S Matchmaker that performs the capability matching. The result of the matching is the advertisement of the providers selected and a reference to the UDDI service record that can be used by the requester to retrieve information from the UDDI registry.

3.3 Related Work

The DAML-S Matchmaker is based on the algorithm described in [26], but in recent years a number of discovery algorithms for Semantic Web services have been proposed. The first of such algorithms, described in [6] , uses the Process

Model of a Web service, expressed using a workflow language, as advertisement of the Web service, and fragments of Process Models as requests. The retrieval mechanism selects from a repository the Process Models that match the request. For the matching to work, the requester and the provider should agree on the names of the different types of processes and how do they relate to each other. To this extent it requires an extensive ontology of processes, and indeed the authors rely on the MIT Process Handbook [17] discussed above. The use of Process Models for matching and the use of extensive ontologies of processes are two striking differences with rispect to the DAML-S Matchmaker. Ultimately, the two approaches re complementary and their use depends on the information that is available at the application, for instance whether the a Process ontology is available or not, and the types of queries that the requester can/wants to express.

A number of matching algorithms that have been proposed rely on Description Logics (DL) and subsumption reasoning. Most prominently [?], [?] and [?]. The first one, [?], assumes an extensive representation of types of Web services to specify the type of service. The matching process is based on the subsumption relation between the advertisement and the request. Similarly to the DAML-S Matchmaker, they define a number of degrees of matching, but they also add a degree of match, intersection, that is based on the search of Web services that are classified in branches of the ontology that are sibling to the classification of the request. On the other hand, [?] and [?] assume an intensive representation of capabilities of Web services that is equivalent to the DAML-S Profiles. [?] provides the semantics for a representation of Web services within a DL and a definition of subsumption for Web services. While they provide the theoretical foundations for a matching process, they do not explicitly define one. A matching process is defined in [?] which uses a modified version of the DAML-S Profile to facilitate the subsumption process, and, as in the DAML-S Matchmaker, they assume multiple degrees of matching. As far as we can see, this algorithm describes a different way to achieve the same results of the DAML-S Matchmaker. The only difference is the use of interesection as an additional degree of match that is supposed to detect when only some features of the request are satisfied by the advertisement. The use of interesection in [?] and [?] is problematic since it may overgeneralize. For instance a request for a provider of least 200 computer parts, may be matched by the advertisement of providers of at least 200 items irrespectively of what those items are.

An intersting matching algorithm has been proposed by [?] who attempts to retrieve the smaller subset of web services that maximizes the achiement of the goal of the request, while providing as many inputs as possible. Because the matching process has such different goals when compared with the DAML-S Matchmaker, it is not really possible to compare the two approaches. Nevertheless, it raises an interesting issue that is rarely addressed in the discovery literature: namely the division of tasks between the discovery registries and the requesters for service. The matching engine proposed by [?] seems to be targeted to a very powerful matching process which can identify the functionalties

of clusters of Web services. On the other hand the DAML-S Matchmaker places more responsability on the requesters to decompose their problems in such as way that they can be solved by multiple requesters.

4 Managing Web Services Interaction

The interaction with the DAML-S/UDDI Matchmaker results in a reference to a Web service that the requester can invoke. The next problem of the requester is to use the Process Model of the provider to interact with it. In this section, we will concentrate on how the requester uses the Process Model to interact with the provider. Specifically, we will discuss the architecture and implementation of the DAML-S Virtual Machine: a general purpose processor for the DAML-S Process Model which allows Web services to interact on the basis of the DAML-S specifications. Furthermore we will show that the DAMS-Virtual Machine is consistent with the execution semantics presented in 2.3.

4.1 Architecture of DAMS-Virtual Machine

The architecture of the DAML-S Virtual Machine is shown in Figure 6. The core of the architecture is represented by three components in the center column: the *Web service Invocation*, the *DAML-S Processor* and the *DAML Inference Engine*[6]. The DAML-S Processor is responsible for "driving" the interaction with the provider. More precisely, the DAML-S Processor derives the sequence of processes to be executed dealing with the intrinsic non-determinism of the DAML-S Process Model on the basis of the rules shown in Table 3. The DAML-S Processor relies on the *Web service invocation* module[7] for the message exchange with the provider. Upon receiving a message, the Web service invocation module extracts the message payload, translates it into DAML, and sends it to the DAML Inference Engine. The DAML Inference Engine is responsible for interpreting the messages received, as well as loading additional ontologies which can help the Web service in its interaction. Furthermore, the DAML Inference Engine is responsible for drawing the consequences of the information that it loads.

The other two columns of the diagram in Figure 3 are also very important. The column on the left shows the information that is downloaded from the Web and how it is used by DAML-S Web services. WSDL is used for Web service invocation, while ontologies and DAML-S specifications of other Web services are loaded in the DAML Inference Engine and used by the DAML-S Processor to make decisions on how to proceed. The column on the right shows the *Reasoning System* which is responsible for what the Web service does. For example, if the Web service provides financial consulting the Agent Reasoning System would contain software that performs financial calculations as well as financial decision

[6] The DAML Inference Engine is based on the Jena RDF/DAML parser [18] and the DAML-Jess-KB[32]

[7] The Web service invocation module is based on Apache's Axis [2] and IBM's WSIF[28].

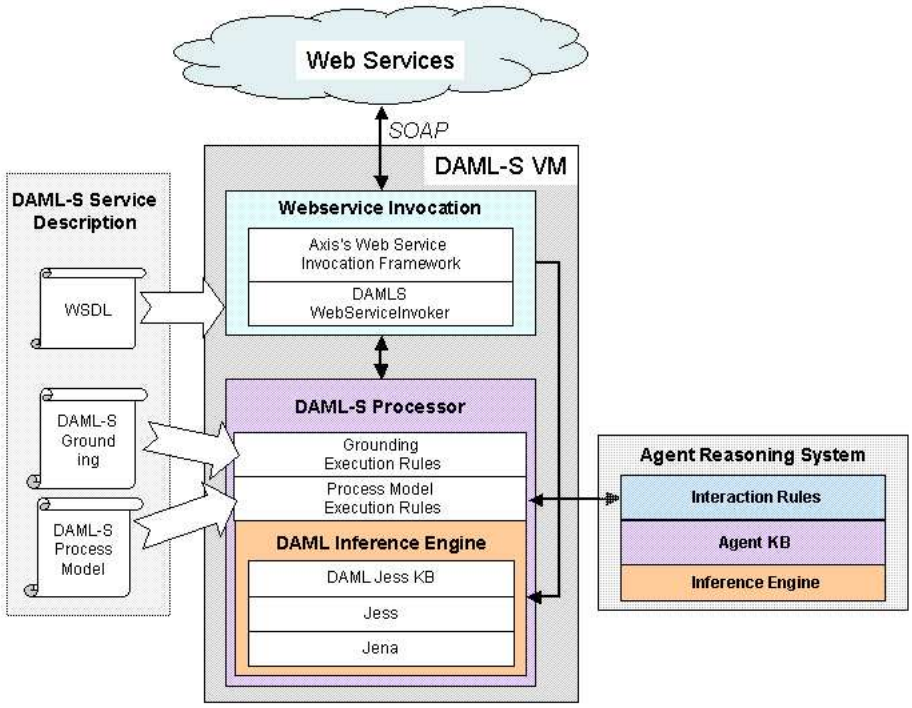

Fig. 6. Architecture of the DS-VM

making such as suggesting stocks to buy. The DAML-S Virtual Machine interacts with the Agent Reasoning System by reporting the information received from the provider or by asking what information to send next.

4.2 Implementation of the Process Model Operational Semantics

The *Process Model Execution Rules* which are employed by the Process Model Processor implement the Process Model operational semantics which we discussed in Section 2.3. In the rest of this section we will discuss the implementation of these rules and we will provide an informal proof of the consistency between the specification and the implementation.

Atomic Atomic processes are implemented by rule (1) in Table 3. Their execution consists of the invocation of the Grounding and ultimately of the operations on the provider. The semantics of executing atomic processes is shown by the rule (FUNC) of Table 1 which consistently with the implementation calls for the execution of an operation on the provider side.

(1) executed (atomic(Process)) ⇐
callGrounding(Process), assert(complete(Process))

(2) executed (sequence(Process,List)) ⇐
executed(first(List)),
executed(sequence(Process,rest(List)))
assert(complete(sequence(Process,List)))

(3) executed (split(Process,List)) ⇐
exec(first(List)), exec(split(Process,rest(List))),
assert(complete(split(Process,List)))

(4) executed (splitJoint(Process,List)) ⇐
exec(first(List)), exec(splitJoint(Process,rest(List))),
complete(first(List)), complete(splitJoint(Process,rest(List)))
assert(complete(splitJoint(Process,List)))

(5) executed (if(Cond,ThenProcess,ElseProcess)) ⇐
(Cond, executed(ThenProcess)) **XOR** executed(ElseProcess)
assert(complete(if(Cond,ThenProcess,ElseProcess)))

(6) executed (choice(Process,List)) ⇐ executed(oneOf(List))
assert(complete(choice(Process,List)))

Table 3. Rules of the Process Model Processor

Sequence Sequences are implemented by rule (2) in Table 3. A sequence of processes is executed by executing the processes in the order established by the sequence. The semantics of a sequence of two processes is shown by the rule (SEQ) in Table 2. A `sequence(Process,List)`, where `List` consists of the processes $p_1, \ldots p_n$ is formalized as follows:[8]

$$\texttt{sequence(Process,}\{p_1, \ldots p_n\}\texttt{)} = \texttt{ do } \{p_1\,\texttt{;}\ldots\texttt{;}\,p_n\}$$

Notice that this is equivalent to the unraveling of

$$\texttt{do } \{p_1\,\texttt{;}\ldots\texttt{;}\,p_n\} \text{ into } p_1 \texttt{ >> do } \{p_2\,\texttt{;}\ldots\texttt{;}\,p_n\}$$

[8] For simplicity, we use the imperative-style do-notation here. as defined in [16]:

$$\texttt{do } \{x \texttt{ <- } e\,\texttt{;}\,s\} = e \texttt{ >>= } \backslash x \texttt{ -> do } \{s\}$$
$$\texttt{do } \{e\,\texttt{;}\,s\} = e \texttt{ >>= } \backslash\_ \texttt{ -> do } \{s\}$$
$$\texttt{do } \{e\} = e$$

where the first process of the list p_1 is evaluated first and then the rest of the list `do` $\{p_2; \ldots; p_n\}$ which shows the consistency between rule (2) and (SEQ).

Split Splits are implemented by rule (3) in Table 3. A split describes the spawn of multiple concurrent computation of processes skipping the wait for their completion. The semantics of sequences is shown by the rule (SPAWN) in Table 2. Formally, a `split(Process,List)`, where `List` consists of the processes $p_1, \ldots p_n$ is expressed as:

$$\texttt{split(Process,}\{p_1, \ldots p_n\}) = \texttt{ do } \{\texttt{spawn } p_1; \ldots; \texttt{spawn } p_n\}$$

As with `sequence`, this is equivalent to launching the first process in the `List` p_1 while concurrently it spawns off $\{p_2, \ldots, p_n\}$ as concurrent processes as it is expressed by rule (3) in Table 3.

SplitJoint SplitJoints are implemented by rule (4) in Table 3. A splitJoint extends split by describing the spawn of multiple concurrent computations of processes with a coordination point at the end of the execution. In our semantics the processes are spawned off sequentially, and the completion of the splitJoint depends on the completion of every process. More formally, the splitJoint construct

$$\texttt{splitJoint(Process,}\{p_1, \ldots, p_n\})$$

is modeled as the following, where each $p_i' =$`do` $\{\ p_i; t\texttt{!done}\}$:

```
do { t <- newport;
     split(Process,{p'_1, ... ,p'_n});
     t?; ... t? }
```

As can be seen, splitJoint behaves like a split with an extra synchronization at the end. The process listens on port t for n messages, where n is the number of sub-processes that were initially spawned. The DAML-S Processor rule (4) in Table 3 for the execution of a splitJoint differs from those for split only in that the splitJoint is complete only when each of the processes it has spawned off is complete. This is clearly equivalent with our semantics, where we first create a port for synchronization, then spawn off the sub-processes and then wait for each of them to send a completion message `done`.

If-Then-Else if-then-else are implemented by rule (5) in Table 3. An if-then-else conditional triggers the execution of the then process when the condition is true, or the else process when the condition is false. The semantics of sequences is shown by the rule (COND-TRUE) in Table 2 and by a symmetrical rule for (COND-FALSE) which is not shown. The if-then-else conditional can be formalized as:

```
if(Cond,ThenProcess,ElseProcess) = (cond Cond ThenProcess ElseProcess)
```

The two XOR conditions in rule (5) of Table 3 correspond to the two rules (COND-TRUE) and (COND-FALSE) of Table 2, which essentially proves the equivalence between rule (5) and (COND-TRUE), (COND-FALSE).

Choice A choice represents a non-deterministic choice among a set of processes which may be forced by the execution context. A choice is executed by executing one of the processes in its list. Choices are implemented by rule (6) in Table 3, while the semantics of the construct is shown by the rule (CHOICE-LEFT) and (CHOICE-RIGHT) of Table 2. A choice construct `choice(Process,List)` where `List` consists of processes $p_1, \ldots, p_n$, is formalized as:

$$\texttt{choice(Process,}\{p_1, \ldots p_n\}) = \texttt{ (choice (choice } p_1 \ p_2) \ \ldots \ p_n)$$

The DAML-S Processor rule (6) in Table 3 for the processing of the choice construct executes one of the set of processes in the choice on the basis of some non-deterministic choice outside its control. This proves the equivalence between rule (6) and (CHOICE-LEFT), (CHOICE-RIGHT).

4.3 The Grounding and the Invocation of Provider

The rules for the Grounding are stored in the *Grounding Execution Rules* module of the DAML-S Processor. These rules allow the compilation of WSDL message structures and the mapping of atomic processes into WSDL operations that can be directly invoked by the *Web service Invocation* module.

In addition, the Grounding rules are used to extract the XSLT [11] transformations that are required when the provider does not express inputs and outputs in DAML. From the implementation point of view, the XSLT transformations are performed by XALAN [3] and then transformed into WSDL messages using JROM [14]. Finally, after the messages are constructed the WSDL operation is invoked using the AXIS framework. Outputs follow the opposite path, the data streams corresponding to the WSDL output messages are returned by the AXIS tools, and fed into JROM and finally transformed into DAML using XALAN. The DAML data is then parsed with the Jena DAML/RDF parser and finally asserted in the Jess KB where they are available for inference and interact with the rest of the knowledge of the Web service.

4.4 Related Work

Using the Petri Net execution semantics for DAML-S proposed in [22], Narayanan et al. have developed a DAML-S interpreter for use with the Petri Net simulation and modeling environment, KarmaSIM. The DAML-S interpreter converts a DAML-S service specification into a Petri Net system, which can then be analysed with the help of KarmaSIM. The KarmaSIM tool and DAML-S interpreter together allow DAML-S services to be simulated interactively and support several kinds of verification and performance analyses of DAML-S service

	Amazon Client	DAML-S Virtual Machine
Average Execution Time	2007 ms	2021 ms
Standard Deviation	1134 ms	776 ms

Table 4. Execution time of Amazon Client and DAML-S Virtual Machine (time in milliseconds)

	DAML-S Virtual Machine	Data Transformation	Invocation
Average Time	83 ms	156 ms	2797ms
Percentage	3%	5%	92 %
Standard Deviation	107 ms	146 ms	1314 ms

Table 5. Distribution of time during the execution

specifications. The DAML-S Virtual Machine does not address verification or performance analyses issues; its focus is instead on dynamic discovery, composition and execution. Once a service has been composed by the DAML-S Virtual Machine, however, the DAML-S interpreter and KarmaSIM tool could be used to analyse the service with respect to its performance and other properties.

5 Performance Evaluation

In the paper so far we demonstrated the correctness of the execution rules used in the DAML-S Virtual Machine. In this section we provide a performance evaluation. We time the DAML-S Virtual Machine during an interaction with the Amazon Web service[9] and we show that the use of DAML-S does not produce a performance penalty.

To estimate the performance of the DAML-S Virtual Machine we performed two experiments: in the first one, we compared the execution time of the DAML-S Virtual Machine with the time required by the client software module provided by Amazon[10] when browsing for a book using the Amazon Web service. This experiment provides the performance cost of using DAML-S and the DAML-S VM. In the second experiment, we provide the average time of the execution of the DAML-S Virtual Machine when both searching and reserving a book. In this experiment we compared the total time the DAML-S Virtual Machine spent in processing DAML-S information with the total time of the interaction with the Amazon Web service. We repeated both experiments at different times of the day to account for the different load conditions both on our side and on Amazon's side. Also, in all experiments we looked exactly for the same items.

[9] We constructed a DAML-S description of the Amazon Web service on the basis of its WSDL description.

[10] Amazon's client requires an input from the user. We hardcoded that input to avoid penalties due to the human interaction.

5.1 Experimental Results

The first experiment was run 98 times over 4 days in varying load conditions. The results of the experiment are shown in Table 4, which shows the average execution time of the Amazon Client and the DAML-S VM and the standard deviation. The results show that the DAML-S Virtual Machine has virtually the same performance of the client distributed by Amazon, with only 14 milliseconds of difference on average.

In the second experiment we computed three measures: the first one is the time required by the DAML-S Virtual Machine to make a decision on the path to take in the Process Model; the second is the time required by the data transformation from DAML to the format required by Amazon, the third is Amazon's invocation time. As in the first experiment we report the average times, and the standard deviation. We also report the percentage of the three averages compared to the total time required by the interaction. The data is shown in Table 5.

Consistently with the first experiment the time required by the DAML-S Virtual Machine is minimal with respect to a call to the Amazon Web site requiring only 3% of the whole interaction time.

The experiments show that the use of the DAML-S Virtual Machine **does not** produce a performance penalty. Indeed the average time required by the DAML-S Virtual Machine for browsing is virtually equivalent to the time required by the Amazon client. This equivalence is explained by the second experiment that shows that the time required by the DAML-S is about 8% of the interaction time, and the majority of that time was required by the XSLT transformations between the XML format required by Amazon and DAML required by the DAML-S Virtual Machine.

6 Using DAML-S for Web Services Composition

So far, we described how the use of DAML-S and the Semantic Web supports capability based discovery, as well as the autonomous invocation of Web services. On their own, these two properties already provide a contribution to the Web services infrastructure. As we have shown, the use of DAML-S contributes capability based discovery to UDDI; and the semantic specification of the information expected by a Web service supports autonomous interaction without the need for direct intervention of human programmers or users.

Discovery and automatic invocation of Web services effectively expands the capabilities of agents that can gain access to Web services. In such case, when solving a goal an agent has two alternatives, either it solves the goal directly using its own capabilities and problem solving, or it subcontracts the goal to some Web service out there that can achieve the same goal. The final solution of the problem is the result of the composition of invocations of Web services and some reasoning that is done autonomously by the agent [21].

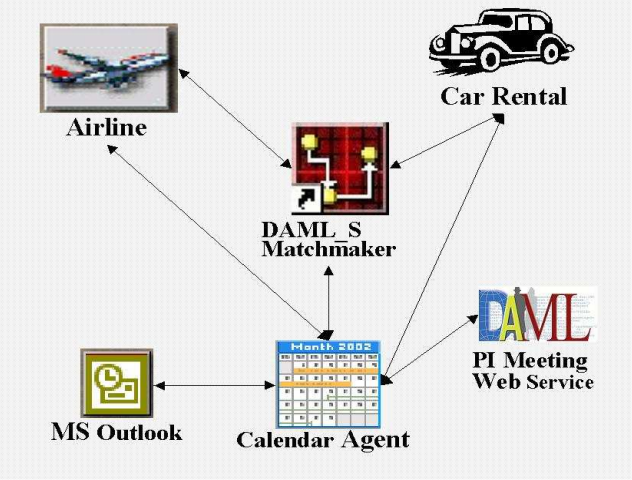

Fig. 7. Description of the system for scheduling a trip to the DAML PI meeting

The Web services composition problem is a new version of the problem of integrating information from different information sources which has been actively investigated in the Data Base community [13] and in the development of Agents and Multi-Agent systems [5, 23]. Web services contribute to this line of research in two ways: first they provide a uniform way to interface information sources; second, the use of the semantic information to represent capabilities and interaction information with Web services provides the basis for a uniform way to specify what information is provided by a Web service, and how to interact with it. In turn this uniform interfaces and descriptions allow existing techniques to scale up to the whole Web.

In the rest of this section, we provide an example of an agent that uses the semantic Web services infrastructure to interact with other Web services only on the basis of DAML-S and information collected from the Semantic Web. Specifically, we employed the RETSINA Calendar Agent (RCal) [27] which can reason about schedules in RDF and DAML and interact with MS Outlook. We expanded the capabilities of RCal with the HiTAP planner [24] which allows the construction of plans involving multiple sources of information. Furthermore, we interfaced RCal with the DAML-S Virtual Machine to allow automatic interaction with other Web services. In our scenario, shown in figure 7, RCal is given the task of organizing a trip to a conference, namely the DAML PI meeting. We assume that the organizers of the meeting publish a Web service which provides information about the meeting, such as time, location, talks, participants and so on. The Calendar Agent verifies the user's availability checking on her schedule stored in MS Outlook, and then uses the DAML-S/UDDI Matchmaker to find airlines, car rental companies and hotels. Finally, RCal uses the DAML-S Virtual Machine to interact with the different Web services until it completes the schedule of the trip which is uploaded into Outlook.

This very simple scenario highlights the challenges of planning for Web services composition. The first challenge was to generate requests for services from goals that RCal could not achieve. For example it had to transform the goal `book(flight)` into a request for an airline which could do the booking of the flight. This was achieved by transforming the goal into a DAML-S Profile whose only output was the information that solves the goal, and empty inputs, preconditions and effects.

The second challenge was to interleave planning and execution. The planning agent had to interact with the Matchmaker, and the Web services themselves before a plan could be constructed. This interaction required the planner to interleave its own planning with the execution of information gathering actions [20, 9]. For this purpose, we adopted HiTAP, an HTN planner that can also suspend planning to execute parts of its plan to extract information that proves to be critical to make choices between alternative plans.

The third challenge was the management of the interaction. The DAML-S Virtual Machine allows the agent to know what messages to send and how to interpret the information it receives, but the agent had to make a decisions about alternatives, and to decide what inputs to send to the providers. The selection of the inputs was based on the inputs of the corresponding step in the plan, while selections between alternative processes in the process model were made by analyzing the consequences of each choice.

Our experiment proved that DAML-S provides the needed information for Web service composition, but it also highlights some of the challenges that have to be explored. Some of these have to do with management of failures of Web services and recovery from those failures. Other problems include interactions between Web services which may undo each other's work.

6.1 Related Work

A service composition prototype has been described by Sirin et al. [29], which enables a user to create a workflow-like service composition by filtering a existing set of services and presenting available service choices to the user at each step. The strength of their system is that it enables step-wise composition of services, where the filtered services presented at each step depend on the services chosen at the previous step and their constraints. This process here is guided heavily by the user, unlike the DAML-S Virtual Machine, which allows an agent to discover services dynamically. The composition prototype could thus be viewed as complementary to the DAML-S Virtual Machine, in that it allows a user to put together services that have been discovered with the help of matchmaking. One issue that remains unclear is how the initial set of discovered services is formed. Crawling the Web for semantic service descriptions, as suggested in the paper, seems practical only for repository services. At this point, a service requestor would anyway need to resort to matchmaking to discover services from the repository, so the problem of matchmaking still needs to be addressed.

The DAML-S Virtual Machine, as far as we know, is the first near-complete system for Web services discovery, composition and execution described in the literature.

7 Conclusions and Future Work

In this paper we presented a vision for a Web of services which combines the growing Web services infrastructure with the Semantic Web. We showed that the excessive reliance of the Web services on pure XML guarantees syntactic interoperability, but it fails to provide semantic interoperability. The result is that Web services may be able to parse the information that they exchange but fail to understand the content of that information.

The Semantic Web has the potential to alleviate and possibly remove this problem by linking the data exchanged to a set of ontologies which specify the conceptual framework that helps with the interpretation of the data. Using these ontologies Web services map the information that they receive to known concepts and then use that mapping to derive consequences of the message.

The task of mapping Web services with the Semantic Web is partly fulfilled by DAML-S: an ontology for the description of Web services. DAML-S provides a way to express capabilities of Web services so that they can be used to discover Web services on the basis of what they do. Furthermore DAML-S provides a way to encode a description of Web services and their interaction protocol.

In this paper we show that DAML-S is not just an abstract description, but that it can be used by programs that implement Web services. Specifically, we show that the capability description can be used by the DAML-S/UDDI Matchmaker to locate the providers on the basis of their capabilities. This is a clear improvement on UDDI that does not provide any capability matching. The second contribution is the DAML-S Virtual Machine which implements the execution semantics of the DAML-S Process Model and can be used to manage the interaction with Web services. Furthermore we demonstrate the use of DAML in two applications and, most importantly, we show that the use of DAML-S does not produce a performance penalty.

As a final note we can ask whether the Semantic Web really delivers on its promises to Web services. While by and large the jury is still out and a definitive answer is still to come, this paper provides an initial answer. Above we showed two approaches to capability matching, and we discussed the differences and the trade-offs. Nevertheless, both representation schemata crucially depend on the existence of ontologies. Indeed, capability matching simply cannot be done without the use of ontologies. The contribution of the Semantic Web to the management of the interaction is a more complicated question which hinges on the contribution of the Semantic Web toward the interpretation of the data exchanged by Web services. The DAML-S Virtual Machine provides an essential step toward the answer of the question, and this will be the main topic of our future research.

8 Acknowledgment

We owe a special thanks to Joseph Giampapa, Frank Huch, Takahiro Kawamura, Takuya Nishimura, Terry Payne and Rahul Singh for their contribution to different parts of this work. The research was funded by the Defense Advanced Research Projects Agency as part of the DARPA Agent Markup Language (DAML) program under Air Force Research Laboratory contract F30601-00-2-0592 to Carnegie Mellon University.

References

1. A. Ankolekar, F. Huch, and K. Sycara. Concurrent execution semantics for DAML-S with subtypes. In *ISWC, 2002*, Sardegna, Italy, 2002.
2. Apache Foundation. Apache - Axis.
3. Apache Foundation. Apache - Xalan.
4. A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/2002/NOTE-wsci-20020808, 2002.
5. G. Barish and C. A. Knoblock. An efficient and expressive language for information gathering on the web. In *Workshop on Is there life after operator sequencing?—AIPS-2002*, pages 5–12, 2002.
6. A. Bernstein and M. Klein. High precision service retrieval. In *ISWC 2002*, Sardegna, Italy, 2002.
7. D. Booth, M. Champion, C. Ferris, F. McCabe, E. Newcomer, and D. Orchard. Web services architecture. http://www.w3.org/TR/2003/WD-ws-arch-20030514/, 14 May 2003. W3C Working Draft.
8. U. C. Bureau. North American Industry Classification System (NAICS). http://www.census.gov/epcd/www/naics.html, 1997.
9. H. Chen, T. Finin, and A. Joshi. Using OWL in a pervasive computing broker. In *Workshop on Ontologies in Open Agent Systems, AAMAS 2003*, 2003.
10. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.
11. J. Clark. XSL Transformations (XSLT) Version 1.0. Technical report, W3C, 1999.
12. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. BPEL4WS white paper. http://www-3.ibm.com/software/solutions/webservices, 2002.
13. H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in tsimmis. In *AAAI Symposium on Information Gathering*, pages 61–64, 1995.
14. IBM Corporation. JROM - Java Record Object Model.
15. N. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):275–306, 1998.
16. S. P. Jones. Tackling the awkward squad: monadic input/output, concurrency, execptions and foreign-language calls. Lecture Notes for a tutorial given at Marktoberdorf Summer School, 2002.

17. T. W. Malone, K. Crowston, B. P. Jintae Lee, C. Dellarocas, G. Wyner, J. Quimby, C. S. Osborn, A. Bernstein, G. Herman, M. Klein, and E. O'Donnell. Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science*, 45(3):425–443, March, 199 1997.
18. B. McBride. Jena: Implementing the RDF model and syntax specification. In *Semantic Web Workshop, WWW2001*, 2001.
19. S. McIlraith and D. Martin. Bringing semantics to web services. *IEEE Intelligent Systems*, 18(1):90–93, 2003.
20. S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, April 2002.
21. S. McIlraith, T. C. Son, and H. Zeng. Semantic web service. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
22. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, Honolulu, Hawaii, USA, May 2002.
23. M. Nodine, W. Bohrer, and A. Ngu. Semantic brokering over dynamic heterogeneous data sources in infosleuth. Technical report, MCC Technical Report, 1998.
24. M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara. A planning component for RETSINA agents. In N. Jennings and Y. Lespérance, editors, *Intelligent Agents VI*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000.
25. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Importing the semantic web in uddi. In *Proceedings of E-Services and the Semantic Web Workshop*, 2002.
26. M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *ISWC2002*, 2002.
27. T. R. Payne, R. Singh, and K. Sycara. Calendar agents on the semantic web. *IEEE Intelligent Systems*, 17(3):84–86, 2002.
28. A. Project). Web services invocation framework. Technical report, Apache Project, 2003.
29. E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure Workshop in conjunction with ICEIS*, 2003.
30. UDDI. The UDDI Technical White Paper. Technical report, OASIS, 2000.
31. W3C. SOAP Version 1.2, W3C Working Draft 17 December 2001. http://www.w3.org/TR/2001/WD-soap12-part0-20011217/, 2001.
32. J. Web and W. Kopena. DAMLJessKB.