The DAML-S Virtual Machine[⋆]

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

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

Abstract. This paper introduces the DAML-S Virtual Machine (DS-VM): an embedded component that uses the DAML-S Process Model to control the interaction between Web services. We provide a proof of the validity of the implementation of the DAML-S Virtual Machine by proving a mapping from the rules used by the DS-VM to the DAML-S Operational Semantics. Finally, we provide an example of use of the DS-VM with a DAML-Sized version of Amazon.com's Web service, and we conclude with an empirical evaluation that shows that the overhead required by the DS-VM during the interaction with Amazon is only a small fraction of the time required by a query to Amazon. The DS-VM provides crucial evidence that DAML-S can be effectively used to manage the interaction between Web Services.

1 Introduction

Web services are emerging as the core technology for e-business transactions. The wide spread use of XML, WSDL and SOAP supports interoperation between Web services, by abstracting implementation details such as programming language and transport protocol that plagued the earlier attempts to achieve distributed computation such as CORBA and Jini. On the other hand, Web services interoperation requires more than abstraction from implementation details, rather Web services should also share the same interpretation of the information that they exchange. This shared interpretation can be achieved only through a semantic description of the information that Web services exchange. Unfortunately, the Web services infrastructure does not provide a semantic layer where the content of the information exchanged by Web services can be expressed. As a consequence, the Web services infrastructure requires that programmers hardcode the interaction between Web services, furthermore the resulting Web services are inherently brittle since they will break whenever the content or the format of their messages change.

DAML-S attempts to overcome the limitations of the Web Services infrastructure by leveraging on DAML+OIL ontologies to provide a semantic specification of what Web services do and the information that they exchange. DAML-S

[⋆] 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.

adopts the prospective that any interaction between Web services involves at least two parties: a provider and a requester, where the requester needs some information or a service that can be provided by the providers. Furthermore, it is the responsibility of the requester to locate the provider and initiate the interaction; while it is the responsibility of the provider to publish a description of its capabilities, or in other words the service that it provides. A typical capability description involves description of how the service is performed, and what information the provider expects from the requester, and finally, how such information is delivered.

The management of the information exchange is specified in the Process Model and Grounding of DAML-S. Operationally, a Process Model is defined as an ordered collection of processes organized on the basis of a workflow, which specifies the sequence of processes performed by the provider during the transaction. Each process is defined as a transformation between an initial state and a final state, where the initial state is specified by the inputs required by the process and the preconditions for the process to run successfully. The final state is described as a set of outputs, or information that results from the execution of the process, and a set of effects that represent physical changes that result from the execution of the process.

During the interaction with the provider, the requester executes the Process Model published by the provider. Each (atomic) process corresponds to an (atomic) information exchange, where the inputs of the process describe the information that the provider expects from the requester, while the outputs correspond to information that the requester will receive in answer from the provider. Ultimately, by following the Process Model, the requester can infer the interaction protocol with the provider.

The contribution of this paper is the description of the DAML-S Virtual Machine (DS-VM) that uses DAML-S descriptions of Web services and DAML ontologies to control the interaction between Web services. The DS-VM described in this paper is one of a kind, because its a complete framework starting from parsing a DAML-S description, executing the process model consistently with the DAML-S operational semantics [2]. Furthermore, the DS-VM uses the DAML-S Grounding to transform the abstract description of the information exchanges between the provider and the requester into WSDL operations and ultimately bits of information that can be exchanged over the net.

To test the DS-VM we generated a DAML- S description of the web service provided by Amazon.com. The result was the automatic generation of a client for the web service that can manage the interaction and automatically interpret the results it obtained from the Web service to select the cheapest book, among all the responses it receives. Ultimately, this corresponds to a net improvement over the Web services infrastructure based on WSDL, because the web service did not require any hand coding of the interaction protocol, as it was automatically derived from the DAML-S description of the web service. Furthermore, by using the Semantic Web to interpret the results, the client was able to draw some inferences and make its selections.

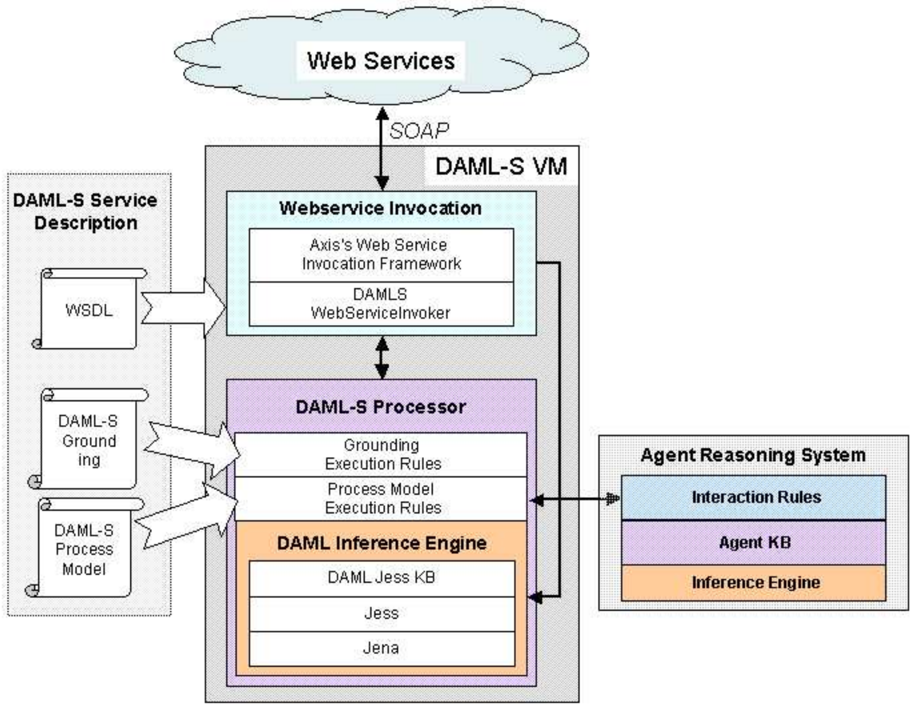

Fig. 1. Architecture of the DS-VM

In the rest of the paper we will discuss in details the theory and the implementation of the DS-VM. In section 2 we describe the architecture of the implementation of the DS-VM. In section 2.2 we show how the implementation maps on the Operational Semantics defined in [2]. In section 3 we describe the generation of the DAML-S description of Amazon.com's Web service; and in section 4 we provide performance measures of the DS-VM and we show that the use of DAML-S is not a computational burden, rather the performance of the DS-VM is equivalent to the performance of the hard coded Web service client provided by Amazon.com and anyway greatly overshadowed by the performance of a HTTP call to the Web service. Finally in section 5 we conclude.

2 Architecture of DS-VM

The architecture of the DS-VM and its relation with the rest of the Web service is described in figure 1. The figure is logically divided in three parts: on the left side the *DAML-S Service Description* specifies the knowledge used by the DS-VM to control the interaction with other Web services. This knowledge is composed by the DAML-S Process Model and Grounding as well as a WSDL description of the bindings. The DS-VM is displayed in the center of the picture. It is logically divided in two modules: the first one is the *DAML-S Processor* which uses the *DAML Inference Engine* and a set of rules implementing the operational semantics of the DAML-S Process Model and Grounding to manage the interaction with the provider. The second component is the *Web service Invocation* module that is responsible for the information transfer with the provider. The last component of the DS-VM is shown on the right side of the figure. It is the *Web service Reasoning System* that is responsible for a number of decisions during the interaction. In the rest of this section we will concentrate on the DS-VM module, and specifically on the DAML-S Processor; we will describe in some details the role of the inference engine and the rules that it adopts to execute the Process Model and Grounding. We will conclude the section by analyzing its relation between the DAML-S Processor and the agent reasoning system, this relation highlights the assumptions that DAML-S makes on the whole Web service.

2.1 DAML-S Processor

The DAML-S Processor is the core of the DS-VM. It is responsible to "drive" the interaction with the provider on the basis of the specification of its Process Model and Grounding. 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. Furthermore, it compiles the inputs to send to the provider and extracts its responses from the outputs.

The DAML-S Processor relies on a DAML inference engine to draw inferences from the DAML-S description of Web services, as well as deriving inferences from the ontologies that it loads. The current implementation is based on the DAML-Jess-KB [9], an implementation of the DAML axiomatic semantics that relies on the Jess theorem prover [6] and the Jena parser [10] to parse ontologies and assert them as new facts in the Jess KB.

In addition to making inferences on the definitions loaded from the Semantic Web, the DAML-S Processor is also responsible to derive the execution of the Process Models and the mappings defined by the Grounding. To perform these inferences, the Process Model uses the rules shown in table 1[1] that implement the execution semantics of of the Process Model, as defined in [3] and [2], which formalize the DAML-S specifications [1].

2.2 Implementation of the Process Model Operational Semantics

The main requirement on the DAML-S Processor is to be faithful to the intended semantics of the Process Model. In this section we review the semantics of each type of process by providing first an informal description, followed by a formal semantics as specified by tables 5 and 6, and finally we will show the mapping into the implementation shown in table 1. The mapping will provide an informal

[1] For ease of explanation the rules are expressed in a Prolog style. In our implementation they are expressed as OPS5 forward chaining rules.

reason of why the rules in table 1 preserve the semantics of the process. Due to lack of space, we omit the full details of the proof. In the rest of this section, we will assume familiarity with the syntax and semantics of *DAML-S Core* [3, 2] and defer a brief review of DAML-S Core to the appendix.

Atomic Atomic processes are executed by invoking the Grounding and information transfer with the provider. The implementation of atomic processes is shown by rule (1) in table 1. The semantics of atomic processes is shown by the rule (FUNC) in table 5 which specifies that p is a symbol that is mapped onto an actual operation $p_{\mathcal{A}}$ over a specified domain of values. The function $p_{\mathcal{A}}$ essentially corresponds to an operation invocation on the Web service. In practice such an invocation results in a call to the grounding as specified by rule (1).

Sequence A sequence of processes is executed by executing the processes in the order established by the sequence. Sequences are implemented by rule (2) in table 1. The semantics of sequences is shown by the rule (SEQ) in table 6 is formalized as follows:[2]

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

Notice that this is equivalent to the unraveling of

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

where the first process of the list p_1 is evaluated first and then the rest of the list `do` $\{p_2\texttt{;}\ldots\texttt{;}p_n\}$ which is exactly what is expressed by rule (2) in table 1.

split A split describes the spawn of multiple concurrent computation of processes skipping the wait for their completion. Splits are implemented by rule (3) in table 1. The semantics of sequences is shown by the rule (SPAWN) in table 6. Formally, a `split(Process,List)`, where `List` consists of the processes $p_1, \ldots p_n$ is expressed as:

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

As with `sequence`, this is equivalent to launching the first process in the `List` p_1 while concurrently the spawning off the remaining processes of the `List` $\{p_2, \ldots, p_n\}$ as concurrent processes as it is expressed by rule (3) in table 1.

[2] For simplicity, we use the imperative-style do-notation here. as defined in [7]:

$$\begin{aligned} \texttt{do } \{x \texttt{ <- } e\texttt{;}s\} &= e \texttt{ >>= } \backslash x \texttt{ -> do } \{s\} \\ \texttt{do } \{e\texttt{;}s\} &= e \texttt{ >>= } \backslash\_ \texttt{ -> do } \{s\} \\ \texttt{do } \{e\} &= e \end{aligned}$$

splitJoint A splitJoint extends split by describing the spawn of multiple concurrent computation of processes with a coordination point at the end of the execution. SplitJoints are implemented by rule (4) in table 1. In our semantics the processes are spawned off sequentially, and the completion of the splitJoint depends on the completion of every process.

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

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

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

This rule is equivalent to the split rule, with the only exception that the spawning process listens on port t for n messages, where n is the number of sub-processes that were initially spawned. Similarly, rule (4) in table 1 used by the DAML-S Processor to control the execution of a splitJoint differs from those for split only in that the splitJoint is complete only when each one of the sub-processes signals its own completion. Such a signal is produced by asserting the completion of the process as the last step of its execution as shown by all the rules in table 1.

if-then-else 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. if-then-else are implemented by rule (5) in table 1. The semantics of sequences is shown by the rule (COND-TRUE) in table 6 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 1 correspond to the two rules (COND-TRUE) and (COND-FALSE) of table 6, which essentially proves the equivalence between the two sets of rules.

choice A choice represents a non-deterministic choice among a set of processes which may be forced by the execution context. The execution of a choice consists of executing one of the processes in its list. Choices are implemented by rule (6) in table 1, while the semantics of the construct is shown by the rule (CHOICE-LEFT) and (CHOICE-RIGHT) of table 6. A choice `choice(Process,List)` where `List` consists of processes $p_1, \ldots, p_n$, is formalized as:

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

The DAML-S Processor rule (6) in table 1 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 and is clearly equivalent to the semantics of the `choice` construct of DAML-S Core.

```
(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 1. Rules of the Process Model Processor

2.3 The Grounding and the Invocation of the Provider

The semantics of atomic processes, as described above, forces the execution of the DAML-S Grounding. The rules for the Grounding are stored in the *Grounding Execution Rules* module of the DAML-S Processor. These rules allow the compilation 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 [5] transformations that are required when the provider does not use DAML as a transmission language. Essentially, these transformations provide a translation from the serialization of the data sent over the network, to the representation of the content of that data. The rationale of the translation is that the information contained by the data types and the information contained by the DAML ontologies should be equivalent. They are just presented in very different ways. The use of XSLT transformations adds a new level of abstraction to DAML-S allowing the description to be specified in terms of ontological classes and instances, whereas the actual data transmission can adopt any arbitrary format. Furthermore, it allows any Web service to be represented by DAML-S independently of its own internal use of DAML.

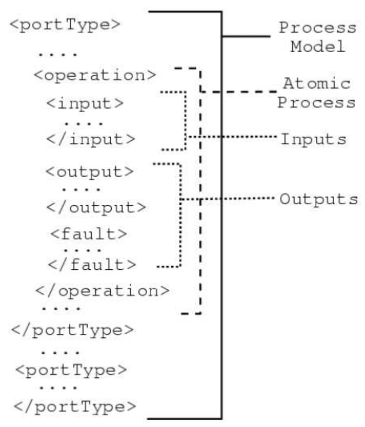

Fig. 2. Rationale of the translation from WSDL to DAML-S

From the implementation point of view, the XSLT transformations required by the Grounding are performed by XALAN [4] and then transformed into WSDL messages using JROM [8]. 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.

2.4 Interaction with the Reasoning System

The semantics discussed in the previous sections is mute on the Web service Reasoning System and concentrates on the specifications of the interaction between Web services. Nevertheless, the Reasoning System is responsible for many of the decisions that have to be made while using DAML-S. For instance, the application level is responsible for the use of the information extracted from the messages received from other Web Services or to decide what information to send to other Web Services. In order to take advantage of the flexibility supported by DAML-S, the Reasoning System should support non-deterministic choices while maintaining efficiency and control on the behavior of the Web Service. In ultimate analysis, DAML-S requires applications that look more like intelligent software agents than traditional e-commerce applications.

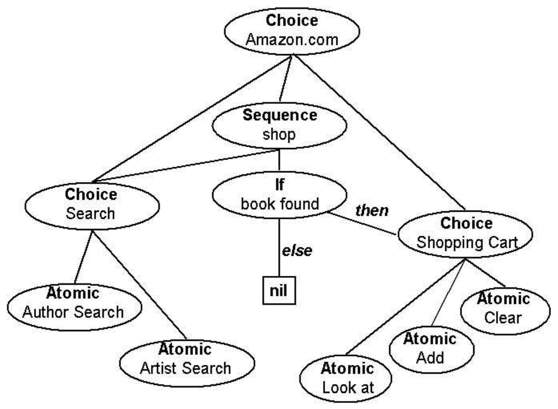

Fig. 3. The Process Model of Amazon.com's Web service

3 Using DAML-S to interact with Amazon.com

Amazon.com provides a widely available Web service[3] which allows users to browse Amazon's databases, locate books and other products and put them in a shopping cart that can be accessed from the Amazon Web site. Through the Web service, users or programs can perform a wide range of keyword searches on the Amazon data bases, for example they are able to search for books with a given author, or look for products of a manufacturer, or search for DVD of movies of a given director. Keyword searches available, span over the whole set of data provided by Amazon, from customer's reviews to Seller profiles.

Amazon provides a SDK to build clients for its Web service, and many tutorials are available to implement the clients using different development kits from Microsoft .Net to IBM's Web Sphere. Amazon also provides a WSDL specification of its Web service that describes operations that can be performed with Amazon, using remote procedure call type of interaction.

We generated a DAML-S specification of the Amazon Web Service generalizing the WSDL description using WSDL2DAMLS [11]. WSDL2DAMLS is a tool that parses the WSDL specification and automatically generates a process model with an atomic process for each WSDL operation, and a complete specification of the grounding and a partial Profile. The translation scheme used by WSDL2DAMLS is shown in figure 2. Furthermore, WSDL2DAMLS generates DAML ontologies by mapping each abstract complex type in the WSDL specification to a DAML class.

Since DAML-S requires more information than the information available in WSDL, for example WSDL does not specify any control relation between the operations that could be invoked in any arbitrary order, the mapping produced

[3] For more information visit http://www.amazon.com/webservices

by WSDL2DAML-S is only a partial, and it still requires the intervention of a programmer to complete the generation of the Process Model and the Profile.

The use of WSDL2DAMLS saved us a great deal of effort in translating every single message type and operation, furthermore it provided us an automatic generation of the Grounding. The only problem with the translation was the generation of the DAML ontology from the complex types, the types used in the WSDL specification are totally arbitrary. While they may have facilitated the work of Amazon's programmers, in general they do not provide an ontological description of books or book selling. For example, the centerpiece of Amazon's Web service type structure is a complex type called *Details* which provides details about every type of product sold by Amazon. Details can be used to describe books, DVDs and other products, with the exception that only some fields apply to books, while others apply only to DVDs and so on. Hence we have to write an XSL stylesheet to transform Amazon's Details complex type to instances of our Ontologies.

As the last step, we completed the process model for the Amazon Web service by adding the control flow among the different operations. The result of this work is shown in figure 3. The main operation of the Process Model is a choice between three types of operations [4]. The first one is *browsing* the Amazon data bases which can be done either by Author search, or by Artist search. The second operation is *shopping* that can be done by first browsing, if anything is found, then adding the item to the shopping cart. The last set of operations allows analysis and modification of the shopping cart by clearing it or adding new items or looking at its content. Finally, we fed the generated Process Model, Grounding and Amazon's WSDL to the DS-VM to invoke the Amazon web service and we were able to perform arbitrary searches for books and CDs using DAML-S as the only specification of the server.

The last problem of interacting with the Amazon Web service has been the implementation of a principled set of rules to make a selection between the different choices in the Process Model. For instance, the Amazon's Process Model allows browsing, shopping or looking at the shopping cart. This type of decisions show that a Web service client has two type of problems: the first one is to decide how to use a Process Model, the second one is to implement those decisions. In our case, the first problem corresponds to deciding whether to browse or shop, the second one is to actually interact with Amazon to browse or shop. We assume that the DS-VM solves only the second problem, while the first one is relegated to the *Agent Reasoning System* shown figure 1, which contains a set of *Interaction Rules* that control the decisions of the agent. Specifically, the rules we implemented take into account the information available to the agent,

[4] Amazon's Web service allows many more types of searches than the set that we implemented. The problem we faced is that all of these searches are keyword based and they do not guarantee the type of the result and it is virtually impossible to know what type of object is returned. We therefore decided in our first attempt to limit to searches that are guaranteed to return books and CDs. Further analysis of the details type may allow us to perform additional types of searches.

and goals it wants to achieve, and then it selects a path in the Process Model that can be executed with only the information available, and that provide the expected results.

4 Performance Measure

In the paper so far we demonstrated the correctness of the execution rules used in the DS-VM and we provided an example of usage of the DS-VM to connect to the Amazon Web service. In this section we provide a performance evaluation of the DS-VM and we show that, at least in the case of interacting with Amazon, the use of the DS-VM does not produce a performance penalty.

To estimate the performance of the DS-VM we performed two experiments. In the first one, which was restricted to the browsing operation, we compared the execution time of the DS-VM with the time required by the client provided by Amazon[5]. In the second experiment we provide the average time of the execution of the DS-VM when both browsing and reserving a book. In this experiment we could not compare with the Amazon client reserving books is beyond what the capabilities of the client that Amazon provides. In the second experiment we then compared the total time the DS-VM spent in processing DAML-S information with the total time of the interaction with the Amazon Web service. We repeated the experiments during different time of the day to account for the different load conditions both on our side and on Amazon's side. Also, in all experiments we requested books from the same author.

4.1 Experimental Results

In the first experiment was run 98 times over 4 days in varying load conditions. The results of the experiment are shown in table 2.

	Amazon Client	DS-VM
Average Execution Time	2007 ms	2021 ms
Standard Deviation	1134 ms	776 ms

Table 2. Execution time of Amazon Client and DS-VM (time in milliseconds)

The first experiment shows that the DS-VM has virtually the same performance of the client distributed by Amazon, with only 14 milliseconds of difference on average.

In the second experiment we analyze how time is distributed in an interaction with Amazon when browsing and reserving a book. We computed three times:

[5] The Amazon's client requires an input from the user, we hardcoded that input to avoid penalties due to the human interaction

first the time required by the DS-VM to make a decision on the path to take in the process model, create the instances of the processes, and executing the processes; the second time is the time required by the data transformation from DAML to the format required by Amazon, and finally the 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 3[6].

	DS-VM	Data Transformation	Invocation
Average Time	83 ms	156 ms	2797ms
Percentage	3%	5%	92 %
Standard Deviation	107 ms	146 ms	1314 ms

Table 3. Distribution of time during the execution

Consistently with the first experiment the time required by the DS-VM 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 DS-VM **does not** produce a performance penalty. Indeed the average time of the required by the DS-VM 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 DS-VM.

5 Conclusions

In this paper we introduced the DAML-S Virtual Machine, which, as far as we know, is the first complete implementation of DAML-S consistent with the operational semantics of the DAML-S Process Model. In the paper we discussed the architecture of the DS-VM and the rules that it uses to regulate the interactions with the providers. Furthermore, we showed that those rules are consistent with the semantics of the Process Model. Finally, we discussed the use of the DS-VM to interact with Amazon's Web service.

This paper provides an opportunity to analyze the contribution of DAML-S and the Semantic Web to the Web services infrastructure. Specifically, three questions emerge: the first one is whether there is any use of ontological information in the Web services representation; the second one is whether DAML-S representation of Web services is adequate or even useful. The last question is whether DAML-S can be effectively used in the interaction with Web services.

[6] The data does not account for the loading of the Process Model and the related ontologies since no similar operation was performed by the Amazon's client

The paper provides an answer to the last question showing that DS-VM is as efficient as the Amazon's own clients, but it also provides an initial answer the previous two questions.

Our experience with Amazon shows that ontological information facilitates the representation and use of Web services. For example, Amazon represents its products using only one data structure that does not distinguish between books, Cd's or electronic devices. That data structure depends on Amazon's internal way to represent products and it does not have any ontological status. As a consequence any client is forced to implement translation rules between Amazon's representation and their own. In practice, the result is an explosion of mappings, one for each provider. The Semantic Web provides a natural *interlingua* that is understood by all clients. As a consequence, using DAML-S, Amazon publishes its own internal mapping from its own data structure to DAML leaving to the clients the task of mapping from DAML to their own data structure. The result is that each client needs only one mapping to the ontology and then it can interact with any book selling web service.

The last question is whether there is a need for DAML-S representation. DAML-S requires the specification of the consequences that result from the execution of a process. This is probably the main contribution of DAML-S representation to the description of Web services. We made an essential use of this feature when our Amazon client had to make a selection between the different activities: look for books vs look for Cd's vs other types of products. To make the selection, the client needs to know what are the consequences of each choice, then match them against its own goals. Ultimately, this is possible only because DAML-S supports the representation of such consequences. Other Web service representation languages such as WSCI and BPEL4WS that do not support the representation input, outputs preconditions and effects of processes executed by Web services force programmers hardcode these choices in the clients leading to overspecialization which leads to brittleness and maintenance problems.

References

1. A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s: Web service description for the semantic web. In *ISWC 2002*, Sardegna, Italy, 2002.
2. A. Ankolekar, F. Huch, and K. Sycara. Concurrent execution semantics for daml-s with subtypes. In *ISWC, 2002*, Sardegna, Italy, 2002.
3. A. Ankolekar, F. Huch, and K. Sycara. Semantics for the web services specification language daml-s. In *Proceedings of the Fifth International Conference on Coordination Models and Languages,, 2002*, 2002.
4. Apache Foundation. XALAN, (http://xml.apache.org/xalan-j/).
5. J. Clark. XSL transformations (XSLT) version 1.0. Technical report, W3C, 1999.
6. E. Friedman-Hill. Jess.
7. S. Haskell and J. Peyton. Tackling the awkward squad: monadic input/output, concurrency, execptions and foreign-language calls. Lecture Notes for a tutorial given at Marktoberdorf Summer School, 2002.

Σ	$\Sigma \subseteq Exp(\Sigma)$
var	$Var^{\tau} \subseteq Exp(\Sigma)^{\tau}$
abs	`\`x `->` $e \in Exp(\Sigma)^{\tau_1 \to \tau_2}$ for $x \in Var^{\tau_1}$, $e \in Exp(\Sigma)^{\tau_2}$
appl	$(e_1\ e_2) \in Exp(\Sigma)^{\tau_2}$ for $e_1 \in Exp(\Sigma)^{\tau_1 \to \tau_2}$, $e_2 \in Exp(\Sigma)^{\tau_1}$
cond	`cond` $e\ e_1\ e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$ for $e \in Exp(\Sigma)^{\texttt{boolean}}$, $e_1, e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
return	`return` $e \in Exp(\Sigma)^{\texttt{IO}\ \tau}$ for $e \in Exp(\Sigma)^{\tau}$
seq	e_1 `>>=` $e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau_2}$ for $e_1 \in Exp(\Sigma)^{\texttt{IO}\ \tau_1}$, $e_2 \in Exp(\Sigma)^{\tau_1 \to \texttt{IO}\ \tau_2}$
send	$e_1 \texttt{!} e_2 \in Exp(\Sigma)^{\texttt{IO ()}}$ for $e_1 \in Exp(\Sigma)^{\texttt{Port}\ \tau}$, $e_2 \in Exp(\Sigma)^{\tau}$
rec	$e\texttt{?} \in Exp(\Sigma)^{\texttt{IO}\ \tau}$ for $e \in Exp(\Sigma)^{\texttt{Port}\ \tau}$
port	$\texttt{newPort}\tau \in Exp(\Sigma)^{\texttt{IO Port}\ \tau}$ for $\tau \in \mathcal{T}$
spawn	`spawn` $e \in Exp(\Sigma)^{\texttt{IO ()}}$ for $e \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
choice	`choice` $e_1\ e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$ for $e_1, e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
serv	$s\ e_1 \cdots e_n \in Exp(\Sigma)^{\tau}$ for $e_i \in Exp(\Sigma)^{\tau_i}$, $s \in \mathcal{S}^{\tau_1 \to \cdots \to \tau_n \to \tau}$

Table 4. DAML-S Core Expressions

8. IBM Corporation. JROM - java record object model.
9. J. Kopena and W. Regli. DAMLJessKB: A tool for reasoning with the semantic web. In *ISWC 2003*, 2003.
10. B. McBride. Jena: Implementing the rdf model and syntax specification. In *Semantic Web Workshop, WWW2001*, 2001.
11. M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura. Towards a semantic choreography of web services: from WSDL to DAML-S. In *ICWS 2003*, Las Vegas, USA, 2003.

A DAML-S Core

In this section, we present a brief review of the syntax and formal semantics of DAML-S Core, as presented in [2].

A.1 Syntax

The set of *DAML-S Core expressions*, $Exp(\Sigma)$, over Σ, which represents a set of symbols for basic operations, is defined in Table 4. Given a type expression τ, the set of expressions of type τ is denoted by $Exp(\Sigma)^{\tau}$.

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

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

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

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

Table 5. Semantics of DAML-S Core - I

A.2 Operational Semantics of DAML-S Core

In a Σ-Interpretation $\mathcal{A} = (A, \alpha)$, A is a T-sorted set of concrete values and α an interpretation function that maps each symbol in Ω, the set of all constructors defined through DAML+OIL, to a function over A.

Definition 1 (State). *A* state *of execution within DAML-S Core is defined as a finite set of agents: State* $:= \mathcal{P}_{fin}(Agent)$ *An* agent *is a pair* (e, φ)*, where* $e \in Exp(\Sigma)$ *is the DAML-S Core expression being evaluated and* φ *is a partial function, mapping port references onto actual ports:*

$$Agent := Exp(\Sigma) \times \{\varphi \mid \varphi : \texttt{PortRef} \longrightarrow \texttt{Port}^{\mathfrak{A}}_{\tau}\}$$

for all τ*, where* $\texttt{Port}^{\mathfrak{A}}_{\tau} := (A^{\tau})^*$ *and* `PortRef` *is an infinite set of globally known unique port references, disjoint with* A*. Since no two agents can have a common port, the domains of their port functions* φ *are also disjoint.*

Definition 2 (Evaluation Context). *The set of* evaluation contexts $\mathcal{EC}$ *for DAML-S Core is defined by the context-free grammar*

$$E := [\,] \mid \phi(v_1, \ldots, v_i, E, e_{i+2}, e_n) \mid (E\ e) \mid (v\ E) \mid E \texttt{ >>= } e$$

for $v \in A$*,* $e, e_1, e_2 \in Exp(\Sigma)$*,* $\phi \in \Omega \cup \mathcal{S} \backslash \{\texttt{spawn}, \texttt{choice}\}$.

Definition 3 (Operational Semantics). *The* operational semantics *of DAML-S is* $\longrightarrow \subset \text{State} \times \text{State}$ *is defined in Tables 5 and 6. For* $(s, s') \in \longrightarrow$*, we write* $s \longrightarrow s'$*, denoting that state* s *can transition into state* s'*.*

The application of a defined service is essentially the same as the application rule, except that the arguments to s must be evaluated before they can be substituted into e. In a [SEQ], if the left-hand side of >>= returns a value v, then v is fed as argument to the expression e on the right-hand side.

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

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

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

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

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

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

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

Table 6. Semantics of DAML-S Core - II

Evaluating `spawn` e results in a new parallel agent being created, which evaluates e and has no ports, thus φ is empty. Creating a new port with port descriptor p involves extending the domain of φ with p and setting its initial value to be the empty word ϵ. The port descriptor p is returned to the creating agent. The evaluation of a receive expression $p?$ retrieves and returns the first value of p. The port descriptor mapping φ is modified to reflect the fact that the first message of p has been extracted. Similarly, the evaluation of a send expression, $p\texttt{!}v$, results in v being appended to the word at p. Since port descriptors are globally unique, there will only be one such p in the system.

The rules for (COND-FALSE) and (CHOICE-RIGHT) are similar to the rules for (COND-TRUE) and (CHOICE-LEFT) given in Table 6. If the condition b evaluates to `False`, then the second argument e_2 is evaluated next, instead of e_1 For a choice expression e_1+e_2, if the expression on the left e_2 can be evaluated, then it is evaluated.However, the choice of which one is evaluated is made non-deterministically.