Stepwise Refinable Service Descriptions: Adapting DAML-S to Staged Service Trading*

Michael Klein, Birgitta König-Ries, and Philipp Obreiter

Institute for Program Structures and Data Organization
Universität Karlsruhe
D-76128 Karlsruhe, Germany
{kleinm,koenig,obreiter}@ipd.uni-karlsruhe.de
http://www.ipd.uni-karlsruhe.de/DIANE/en

Abstract. In order for service-oriented architectures to become successful, powerful mechanisms are needed that allow service requestors to find service offerers that are able to provide the services they need. Typically, this service trading needs to be executed in several stages as the offer descriptions are not complete in most cases and different parameters have to be supplemented by the service requestor and offerer alternately. Unfortunately, existing service description languages (like DAML-S) treat service discovery as a one shot activity rather than as a process and accordingly do not support this stepwise refinement. Therefore, in this paper, we introduce the concept of partially instantiated service descriptions containing different types of variables which are instantiated successively, thereby mirroring the progress in a trading process. Moreover, we present possibilities how to integrate these concepts into DAML-S syntactically.

1 Introduction

In distributed environments, services offer an important possibility to enable cooperation among the participating devices. On the one hand, members can offer their resources as services and on the other hand, they can use the functionalities offered by other members in order to enable complex applications. When regarding typical distributed service-oriented systems like internet-based web services or services in peer-to-peer or ad hoc networks, we notice what they have in common is the fact that the participating devices are loosely coupled, only. In these environments, like on a public marketplace, services are traded, i.e. service offerers publish a description of their service, which are in turn searched by potential service users.

At first glance, this trading seems to be comparable to "normal" internet searches, e.g., for certain documents. However, when one takes a closer look, it becomes obvious that indeed this is *not* the case. Typically, service offers are

* This work is partially funded by the German Research Community (DFG) in context of the priority program (SPP) no. 1140.

M.E. Orlowska et al. (Eds.): ICSOC 2003, LNCS 2910, pp. 178–193, 2003.

not fixed descriptions of all execution details, but contain different categories of variables which have to be instantiated in subsequent steps of the service negotiation before the service can be executed properly. Consider, e.g., a printing service: The description has to be made in a generic way, i.e. leaving space for a user to specify the resolution and color type of the printout as well as the document he wants to have printed. Moreover, the service offerer cannot provide quality of service parameters like the time when printing will finish without knowing the size and format of the document[1].

Unfortunately, existing service description languages do not support incomplete service descriptions which can be successively completed in further steps. Therefore, in this paper, we present a novel approach to service trading which takes into consideration that service trading is an interactive process rather than a one-shot activity. The approach is mainly based on DAML-S [1] and additional service ontologies that have been developed in our research project DIANE (see [2,3]).

In Section 2, we will first analyze existing service description languages and shoe that most of them are not capable of accurately describing a configurable service. Therefore, we will examine typical stages of service trading in Section 3 using the printer example from above. Based on these deliberations, we will derive a generic state oriented service description language which describes the initial and resulting state of the service including fixed and variable parts. Each of these variable parts is expressed by a variable of a certain category exactly defining who has to instantiate it and when. It will turn out that the variables will not always be freely instantiatable and independent of one another. Therefore, we formalize instantiation restrictions for these variables in a next step in Section 4. Finally, in Section 5, we show how the approach could be integrated into DAML-S using a special valueless RDF construct. The papers ends with a conclusion and an outlook to future work in Section 6.

2 Related Work

In this section, we want to examine existing service description languages. Besides languages that are explicitly denoted to describe *services*, we also want to take a look at languages to describe methods and operation of *components* in a middleware environment (sometimes also called *component description languages*). Typically, these component services have a finer granularity than common web services, are often developed for a non-human use, and seldom implement a self-contained action (see [4]). In the following, we will refer to these component operations as *services*, too, and also denote their descriptions as *service descriptions*.

The most frequently used languages are **message oriented service description languages**. They try to describe a service by explaining the values that are entering and leaving the black box service (see Figure 1a). If the service is realized by methods of an object, this can be achieved by exporting the

[1] We will take a more detailed look at this printer example in Section 3.2.

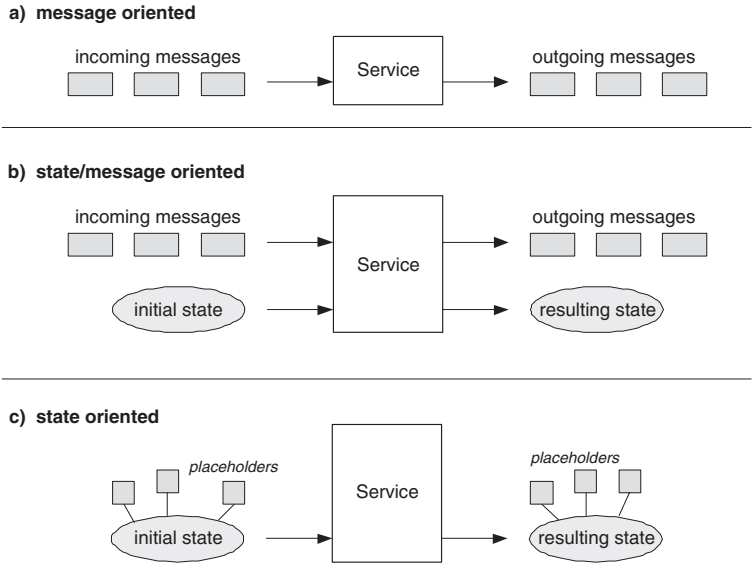

Fig. 1. Different types of service description languages: (a) Message oriented languages try to describe services by their incoming and outgoing data, (b) state/message oriented languages additionally express the functional semantics by describing the initial and resulting state, whereas (c) purely state oriented languages also omit the messages in favor of configurable placeholders in the state description.

interface of the class, which defines the ingoing method parameters as well as the outgoing method result. Well known *interface definition languages* (IDL) are used in CORBA (as OMG IDL [5] or Java IDL [6]), Microsoft's RPC, COM and DCOM (as Microsoft IDL or MIDL [7]), in the area of web services (as Web IDL [8]) and also Java RMI [9], Jini [10] or Enterprise Java Beans [11] (as standard Java interfaces). On the other hand, languages like WSDL [12], e-Speak (used within Hewlett Packard's Web Services Platform [13]), the Collaboration Protocol Profile from ebXML [14] and IBM's Network Accessible Service Specification Language (NASSL) [15] more clearly point out the messages that enter and leave the service.

However, these message centered description languages suffer from one severe drawback: The semantics of the service is left open, i.e. the description leaves unspecified how inputs and outputs are connected and what side effects are performed. This functional semantics can only be guessed by a human user from the operation's name or a textual description. Because of their severe problems with respect to automatic service trading, we will not consider these languages any further.

This drawback is removed in **state/message oriented service description languages**. In addition to the flow of information, they try to capture the functional semantics by describing the state of the world before and after suc-

cessful service execution (see Figure 1b). This can be achieved by the use of state ontologies like in DAML-S [1] (with extensions from [3]), OWL-S [16], and the Interagent Communication Language (ICL) from the Open Agent Architecture [17].

However, this mixed description consisting of message parts and state parts is problematic: First, the influence of the parameters stemming from the exchanged messages on the involved states is left open and therefore unclear[2]. Second, it is unclear when the messages between service requestor and executor have to be sent. Generally, the problem arises from the fact that in these descriptions the abstract service representation is intermixed with a concrete execution realization (i.e. by sending messages). On the other hand, the necessary description of the variable parts of the state is removed from their description and can only be guessed in the description of the exchanged messages. Other approaches try to capture the transition of states more accurately with the help of high level logical programming languages like Golog [18] or Abstract State Machines [19]. However, this leads to an non-declarative, but imperative description of the service action, which prevents a useful comparison with a service request.

As a consequence, the description of messages should be avoided in service description all together, which leads to purely **state oriented service description languages**. Here, the description of the states contains fixed parts, i.e. parts that are completely defined before service trading starts, as well as variable parts, i.e. placeholders which represent values that have to be negotiated before service execution can start (see Figure 1c). Similar ideas are included in LARKS [20], a language to describe agent capabilities. However, the process of correctly filling these values remains unclear. It turns out that each placeholder should be tagged with a label denoting by whom, when and how it should be filled in order to more clearly specify the process that is necessary to configure and execute this service. Nevertheless, the technical details of this configuration process (like the format of exchanged messages etc.) should be left open as they can be derived from the placeholders automatically. As the goal of this paper is to find a service description language for staged service trading, we will introduce a generic state oriented service description language by introducing the concept of states that are configurable with the help of variables and show how these ideas could be syntactically included into DAML-S.

3 The Process

As pointed out above, in contrast to simple internet or information retrieval searches, service trading is a complex, interactive process. In this section, we take a closer look at this process, its different stages, and their respective requirements.

[2] In DAML-S/OWL-S, these connections have to be explicitly listed in a separat part of the description, the so called *Service Model*, which normally is not taken into consideration when matching services within the phase of service trading.

3.1 General Trading Process and Variable Categories

In order to use a service in a loosely coupled, distributed environment, services have to be explicitly announced by their offerer (Stage 1). After that, from a client's point of view, the following three phases (consisting of six stages) can be noticed:

Phase I – Search
Sending a service request (Stage 2) and gathering service advertisements (Stage 3).
Phase II – Estimate
Requesting (Stage 4) and comparing estimates (Stage 5).
Phase III – Execution
Choosing a service (Stage 6) and receiving its results (Stage 7).

Typically, the service advertisements gathered in Phase I are not yet complete as different kinds of parameters are still missing. Therefore, in Phases II and III, the advertisements are successively specialized by instantiating these variables. We denote variables that are used within Phase II *estimate variables* and mark them with index e. Variables used in Phase III are called *execution variables* and are marked with an x. Furthermore, we distinguish variables based on the party that has to instantiate it. Variables that need to be instantiated by the client are called `IN` variables, the ones that are instantiated by the server are called `OUT` variables. To sum up, we differentiate between four *categories* of variables: $\texttt{IN}_e$, $\texttt{OUT}_e$, $\texttt{IN}_x$, and $\texttt{OUT}_x$. Consider as an example the printing service mentioned in the introduction. `IN` variables could be the document identifier and the desired resolution. While the latter needs to be specified during Phase II to allow for e.g. cost estimates, it is sufficient to provide the former in time for service execution, that is in Phase III. Examples for `OUT` variables are the printer location and the estimated completion time. Again, the latter needs to be instantiated for Phase II, the former is known at execution time, only.

3.2 Details of the Process

With the help of the variables introduced above, we can now take a more detailed look at the process of service usage. As identified above, we have seven stages:

(1) Service Announcement
Whenever a device wishes to make a service available to other members of the network, it needs to provide a service advertisement describing the service it is willing to offer. This service advertisement has then to be made known to potential service requestors. This is done by some kind of service discovery mechanism and is outside of the scope of this paper (see [21] for an overview). In [3], we have described a process and a tool to develop such service descriptions. In order for service descriptions to be of any use in a loosely coupled environment, an

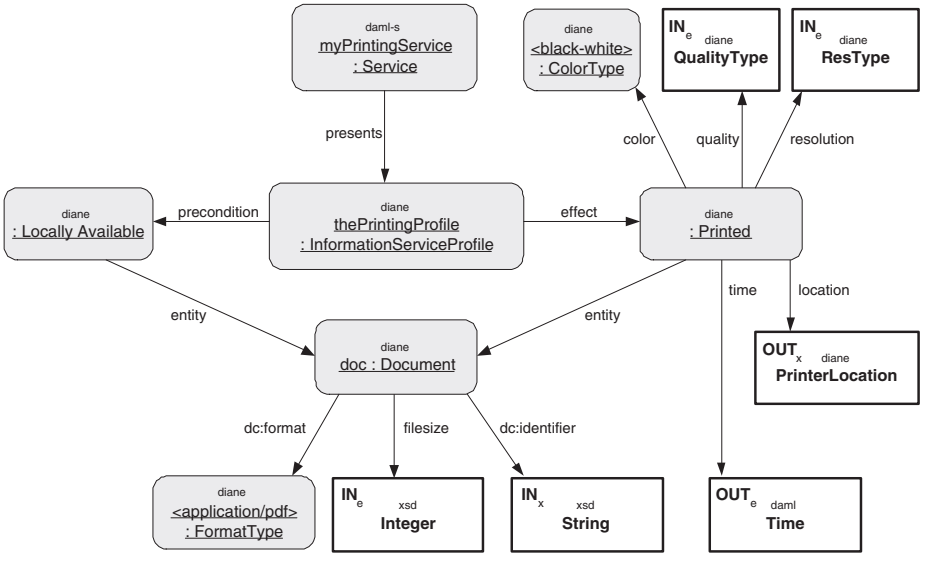

Fig. 2. Advertisement for a printing service.

ontology based approach is needed. In our approach, we distinguish three layers of service description. The top-layer contains the general structure of service descriptions. An example for such a top layer description is a modified DAML-S profile. Here, services are described by the initial states (preconditions) and the resulting states (effects). The middle layer specializes these general state descriptions for different service categories, e.g., information services, shopping services and so on. The specialization is achieved by restricting the types and cardinalities of the service's states. Finally, the third layer contains a collection of domain ontologies. A concrete service description is an instantiation of these three layers using one service category and one or more domain ontologies. However, there are certain aspects of a service (or more precisely the involved states) that cannot be completely instantiated at the time of service description. For these, the four categories of variables introduced above are used.

Consider as an example the advertisement for the printing service depicted in Figure 2. Generally, the service transforms the state of a document from LocallyAvailable (the precondition) to Printed (the effect). Notice that precondition and effect are connected via a common document instance. As mentioned above, some of the document's and state's attributes are already known and instantiated, others are undefined and therefore represented by variables. For example, as the service only allows to print PDF documents, the format of the document is already set. Also the color of the printout is defined: it will be black-white. Other values like the document's file size and its identifier as well

as the printout's quality, resolution, finishing time, and output location[3] are unknown yet and will be filled in the subsequent steps.

(2) Service Search

If a client wants to use a certain service, it first has to find it. Therefore, it creates a *service request* containing at least the wished effects and wished outputs that should be provided by a suitable service. In some cases, also inputs and preconditions can be included in the request showing the initial position of the client before service execution. Like service advertisements, requests consist of instantiated and undefined parts. These undefined parts are also represented by variables (so called *placeholders*) whose instantiation can be restricted to a certain set of values by specially denoted properties. In Section 4, we will examine these restrictions in more detail. This request is dismissed to the discovery layer in order to find service advertisements that could be interesting for the client.

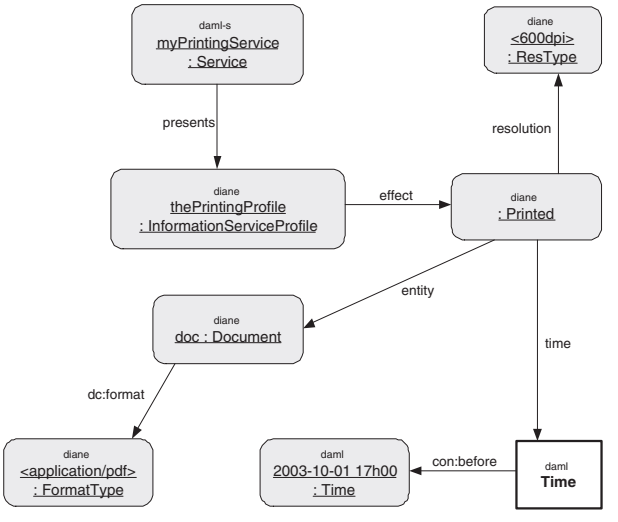

Fig. 3. Service request: Wants a PDF document to be printed in 600 dpi.

Figure 3 shows such a request. A user wants a PDF document to be printed in 600 dpi, which he specifies as effect of the requested service. The fixed values (600 dpi and PDF) are given as instantiated parts. On the other hand, the user does not want to restrict the finishing time of the output to an exact point in time, but wants to allow all times before a certain deadline. Therefore, he uses a placeholder of type Time that represents the finishing time and is restricted to values before 2003-10-01 17h00 by the special property con:before. Besides this

[3] This shows that this printing service automatically and dynamically chooses a device from a pool of physically distributed printers.

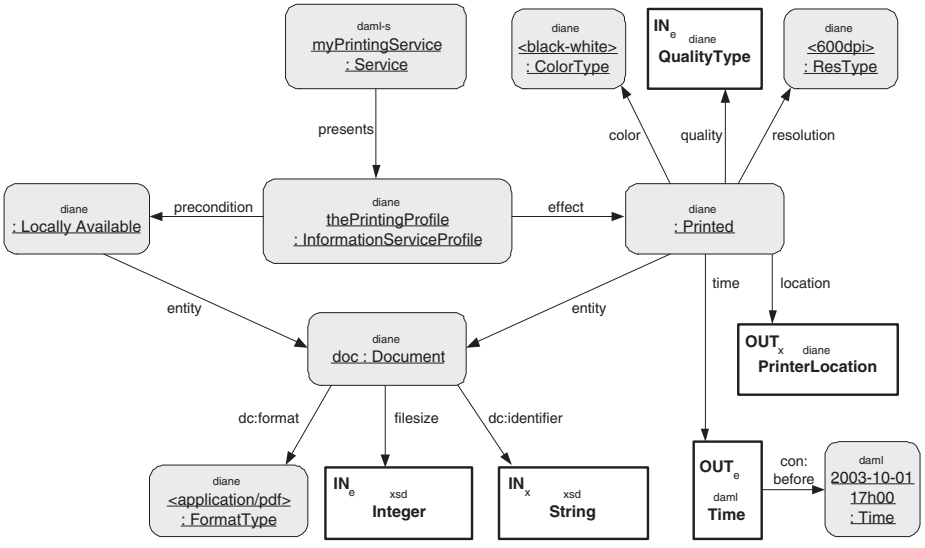

Fig. 4. Estimate Configuration Form.

condition, other instantiation restrictions are thinkable. In the following, we will denote them with the namespace con.

(3) Service Matching

When the discovery layer receives a request, it determines *possibly matching* advertisements for it. Generally, an advertisement possibly matches a request, if (a) all wished effect instances of the request can also be found in the advertisement and (b) there could be a variable binding so that the request and the advertisement are not contradictory. All possibly matching advertisements that can be found are further instantiated according to the restrictions from the requestor and sent back to it as an *estimate configuration form*.

In our example, the service request from Figure 3 possibly matches the advertisement from Figure 2 because the effect Printed matches and points to a PDF document in both cases. Moreover, the finishing time could match but cannot be determined exactly at this stage of service trading. It needs additional information like the concrete value for the $\mathtt{IN}_e$ variable standing for the document size to estimate the ending time. Therefore, this advertisement is a candidate which is further instantiated to the estimate configuration form from Figure 4. It differs from the service advertisements only in two places: the printout's resolution is now fixedly set to 600 dpi (as requested by the user) and the finishing time is restricted to values before a certain deadline with the condition con:before (also requested by the user).

(4) Estimate Configuration

In this stage, the service requestor collects the found service advertisements and filters out the ones that are not interesting for him (for instance because of a

precondition that cannot be provided by the client). After that, he instantiates the missing $\texttt{IN}_e$ variables (i.e., he fills out the estimate configuration form) and sends this *filled estimate configuration form* back to the offerer to provide him with the information he needs for calculating an estimate. Notice that some of the $\texttt{OUT}_e$ variables could also be computed on the client side if the calculation formula was part of the service description.

Figure 5 shows an extract of this filled estimate configuration form. The user has entered the values for the $\texttt{IN}_e$ variables filesize and quality (not shown in the Figure). To calculate the estimated finishing time, this form is sent back to the offerer.

(5) Estimate Calculation

When the service offerer receives a filled estimate configuration form, it computes the complete estimate by filling out the missing $\texttt{OUT}_e$ values. Typically, this computation takes into account (a) the parameters of the client (like the document's size or the wished quality of the printout in our example) which are specified in the estimate configuration form and (b) the current state of the service offerer (like for example the current length of the printing queue). If these values fulfill possibly attached condition properties, this *service estimate* is sent back to the service requestor who also understands it as an *execution configuration form.*

Figure 6 shows the estimate for our example printing service computed from the values in the estimate configuration form. The only change is the value for the finishing time, which has been set to 2003-10-01 16h51 by the service offerer. As it is conform to the condition con:before, it is sent back to the requestor.

(6) Execution Configuration

In this stage, the service requestor collects the estimates and selects one of it. Typically, the main criteria for the selection are the values of the variables that had been placeholders in the service request. As these values have not been specified explicitly, the incoming estimates generally differ in these values and get comparable by them as a result. The chosen estimate serves as execution configuration form at the same time. The client specifies the $\texttt{IN}_x$ variables that are necessary for a proper execution of the service. Often, $\texttt{IN}_x$ variables are filled with values that should be known to the actual service executor only, in contrast

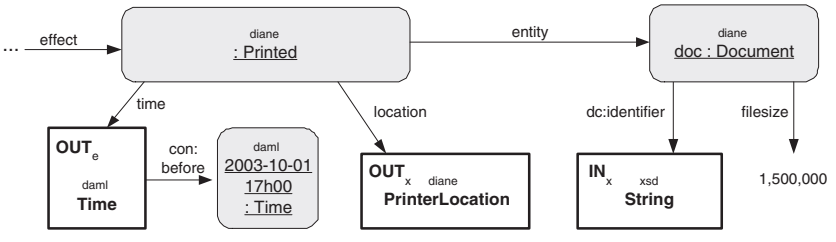

Fig. 5. Part of the Filled Estimate Configuration Form.

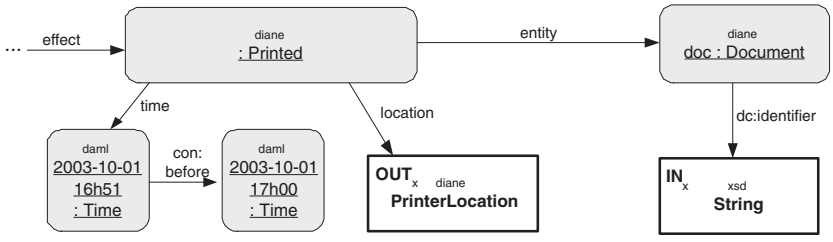

Fig. 6. Service estimate = Execution Configuration Form.

to $\texttt{IN}_e$ variables, which can be freely disseminated to all possible service offerers. Then, the *filled execution configuration form* is sent back to the chosen service offerer.

In our printing example, the client has specified the $\texttt{IN}_x$ variable dc:identifier, i.e. the location of the document to be printed (see Figure 7). Notice that for privacy reasons this URL should not be disseminated to every service offerer, especially as it is not necessary for service search and estimate calculation.

(7) Result Generation
The last step occurs after service execution. Then, the service executor instantiates the $\texttt{OUT}_x$ variables. These contain values that can be determined not until the service execution starts. The resulting *service receipt* is sent back to the service requestor. In the case of our printing service, as soon as the executing printer is determined, the offerer fills in the location of this so that the user can fetch its printout there.

In Figure 8, the service executor has inserted the location of the printout as <room335>. The service description is now fully instantiated as it contains no variables anymore.

3.3 Conclusion

To sum up, service trading does not only consist of sending out a service request and picking the best from a list of received service offers, but the service offers

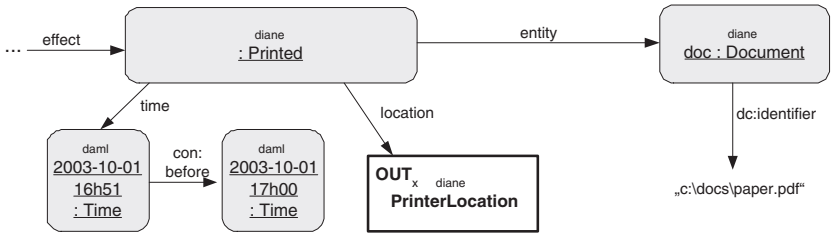

Fig. 7. Filled Execution Configuration Form.

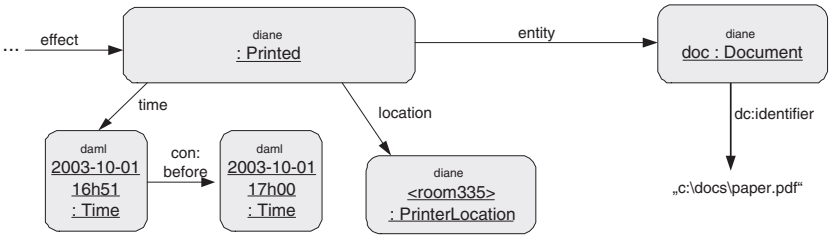

Fig. 8. Service Receipt.

(or more clearly the states involved in them) are developing in several steps. At the beginning, the descriptions contain different categories of variables, which are instantiated in each of the trading stages. Finally, the descriptions (and its states) are completely specified. Notice, that the realization of this instantiation process is explicitly left open. Typically, it is accomplished by sending messages containing the values of the instantiated variables.

4 Instantiation Restrictions on Variables

When a variable is instantiated, in principle, each value from its domain is legal and can be chosen. Often, this behavior is not desirable: (1) A service offerer might want to express that his service only allows, supports or provides certain values for the parameters. In the printer example, only resolutions from 150 to 300 dpi could be offered or the location of the printout could be restricted to rooms of the university. (2) A service requestor might want to express that placeholders in his request should only match certain values in advertisements. In the example in Figure 3, the ending time of the printout is restricted to time values before 2003-10-01 17h00.

Generally, two types of instantiation restrictions can be distinguished:

– **Restrictions concerning a single variable.** Restrictions of this type narrow the instantiation possibilities of one variable independently from other

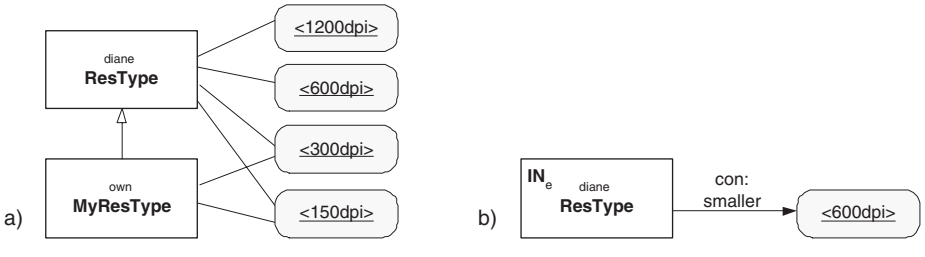

Fig. 9. Technical possibilities to restrict the instantiation of a single, independent variable.

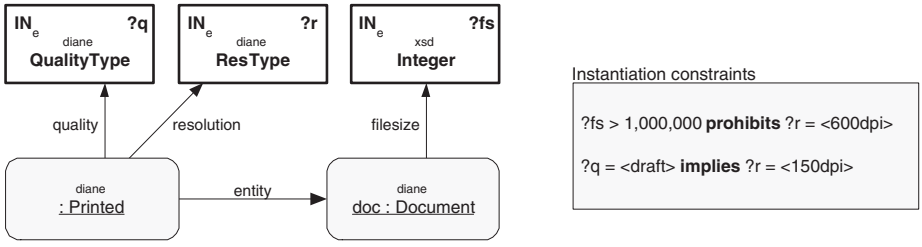

Fig. 10. Restricting several non-orthogonal variables by introducing variable names and an external dependencies list.

variables. Technically, this can be done with two different methods: (1) Introducing own user-defined classes and using them as type for the variable. This new class should be a subclass of the original class and narrow its domain. An example is depicted in Figure 9a. Here, the superclass ResType is an enumeration type describing general printer resolutions. It comprises the values <1200dpi>, <600dpi>, <300dpi>, and <150dpi>. The specialized subclass MyResType is also an enumeration type, but comprises only <300dpi> and <150dpi>. In a service description of a low resolution printer, MyResType could be used as class for the `IN` variable representing the resolution. (2) Inserting special properties that restrict the instantiation of the appropriate variable. In Figure 9b, the `IN` variable of type ResType is restricted with the property con:smaller to resolutions that are smaller than <600dpi>. Notice that these restricting properties can be recognized by their prefix con for constraint.

– **Restrictions concerning several variables.** Restrictions of this type limit the instantiation possibility of one variable with respect to the instantiation of another variable. This becomes necessary if the parameters of a service are not orthogonal, but partially exclude each other. In our printer example, the resolution variable could only be instantiable to <600dpi> if the file size is smaller than 1 MB. Moreover, the printing quality <draft> could always lead to a resolution of <150dpi>. As these restrictions can become very complex (by containing mathematical and logical formulae), they should be separated from the service description graph and collected in an additional constraint list[4]. As an additional requirement, variables with external instantiation dependencies need to have a unique name. Figure 10 shows a possibility to describe the above-mentioned dependencies of our printing service.

5 Syntactical Integration into DAML-S

The most important representative of ontology based service description languages is DAML-S (and its OWL-based pendant OWL-S) [1,16]. In the follow-

[4] Another possibility in RDF based service description would be reification.

ing, we will concentrate on DAML-S together with the state ontologies from [3]. We will examine how to enhance it with the possibility to express incomplete service descriptions, which can be completed successively. As pointed out in the previous section, this can be achieved by using variables. Unfortunately, the concept of variables is not provided in DAML-S and the standards it is based on: DAML, RDFS, RDF, XML Schema, and XML. Therefore, in this section, we propose a syntactical construct for enriching DAML-S based service descriptions (and also other DAML instance graphs) with variables.

Before presenting the construct, we want to explain some important requirements for it: First, as service descriptions are instance graphs, variables should also be regardable as instances of their domain without having been assigned a concrete value yet. This would avoid a mixture of classes and instances, which often leads to unclear semantics. Second, the construct needs to offer the possibility to express the name, the type, the category (i.e. $\texttt{IN}_e$, $\texttt{OUT}_e$ etc.), and possible instantiation constraints of the variable. When looking back to the variables used in the printing example, we observe that only variables of primitive XML schema datatypes (like `xsd:integer` or `xsd:string`, see [22]) as well as DAML enumeration types (like `diane:QualityType`) have been used. Therefore, we will concentrate on these two groups in the following, leaving out variables for general object types (which are not necessary in general as they can be composed of other types) and collection types (which we will have to examine more closely in future).

The construct we propose satisfies these requirements. It is based on the fact that the creation of a new DAML instance can be expressed indirectly with the RDF constructs `rdf:Description` and `rdf:ID`. Therefore, we can create a instance as follows:

```
<rdf:Description rdf:ID="filesizevalue"/>
```

Generally, the type of an instance can be denoted with `rdf:type`. As XML schema datatypes are accessible within DAML via the class `daml:Datatype`, we could specify an integer instance as follows:

```
<rdf:Description rdf:ID="filesizevalue">
  <rdf:type>
    <daml:Datatype rdf:about="xsd:integer"/>
  </rdf:type>
</rdf:Description>
```

Notice that this is an instance of an integer, but its value has not been specified yet. Therefore, this construct can be regarded as a variable and its value could be added in a later stage by inserting an `rdf:value` statement:

```
<rdf:Description rdf:ID="filesizevalue">
  <rdf:type>
    <daml:Datatype rdf:about="xsd:integer"/>
  </rdf:type>
```

```
  <rdf:value>1500000</rdf:value>
</rdf:Description>
```

To be able to express the category of such a variable, we introduce a new class and a new property:

```
<daml:Class rdf:ID="VariableCategory">
  <daml:oneOf rdf:parseType="daml:Collection">
    <Thing rdf:resource="#INe"/>
    <Thing rdf:resource="#OUTe"/>
    <Thing rdf:resource="#INx"/>
    <Thing rdf:resource="#OUTx"/>
  </daml:oneOf>
</daml:Class>

<daml:ObjectProperty rdf:ID="varCat">
  <daml:range rdf:resource="#VariableCategory"/>
</daml:ObjectProperty>
```

This allows us to express that `filesizevalue` is an $\mathtt{IN}_e$ variable:

```
<rdf:Description rdf:ID="filesizevalue">
  <rdf:type>
    <daml:Datatype rdf:about="xsd:integer"/>
  </rdf:type>
  <diane:varCat rdf:resource="#INe"/>
</rdf:Description>
```

Figure 11 shows a complete example by expressing the diagram from Figure 10 in DAML using the above-mentioned constructs. Besides XML schema datatype variables also variables with enumeration types are presented. Notice that the constraint restrictions concerning a single variable can be simply inserted into the description as DAML construct whereas restrictions concerning several variables are written down in a external file.

6 Conclusion and Future Work

In this paper, we have analyzed a typical process of service trading, which generally does not consist of a simple request/response step only, but more interactively extents to several stages. As the existing message and state/message oriented service description are not capable of accurately describing a configurable service, we have presented a generic state oriented service description, which describes the initial and resulting state of the service including fixed and variable parts. Each of these variable parts is labelled showing who should instantiate it, and in which stage and under what restrictions. However, the description explicitly lacks a concrete explanation of the exchanged messages as this can be derived

```
(01) <diane:Printed>
(02)   <diane:quality>
(03)     <rdf:Description rdf:ID="?q"/>
(04)       <rdf:type>
(05)         <daml:Class rdf:about="diane:QualityType"/>
(06)       <rdf:type>
(07)       <diane:varCat>
(08)         <diane:VariableCategory rdf:resource="#INe"/>
(09)       </diane:varCat>
(10)     </rdf:Description>
(11)   </diane:quality>
(12)   <diane:resolution>
(13)     <rdf:Description rdf:ID="?r"/>
(14)       <rdf:type>
(15)         <daml:Class rdf:about="diane:ResType"/>
(16)       <rdf:type>
(17)       <diane:varCat>
(18)         <diane:VariableCategory rdf:resource="#INe"/>
(19)       </diane:varCat>
(20)     </rdf:Description>
(21)   </diane:resolution>
(22)   <diane:entity>
(23)     <diane:Document rdf:ID="doc">
(24)       <diane:filesize>
(25)         <rdf:Description rdf:ID="?fs">
(26)           <rdf:type>
(27)             <daml:Datatype rdf:about="xsd:integer"/>
(28)           </rdf:type>
(29)           <diane:varCat>
(30)             <diane:VariableCategory rdf:resource="#INe"/>
(31)           </diane:varCat>
(32)         </rdf:Description>
(33)       </diane:filesize>
(34)     </diane:Document>
(35)   </diane:entity>
(36) </diane:Printed>
```

Fig. 11. DAML based description for the diagram from Figure 10.

automatically. Finally, we have presented a construct that allows a syntactical integration of the concepts into the service description language DAML-S.

In the future, we will examine possibilities how to automatically construct graphical user interfaces from such a configurable service description. Moreover, more complex dependencies between variables (e.g. those including time) and dependencies between several services (e.g. those between a storage and a retrieval service) will be analyzed.

References

1. Defense Advanced Research Projects Agency: DARPA agents markup language - services (DAML-S). (http://www.daml.org/services/)
2. Institute for Program Structures and Data Organization, Universität Karlsruhe: DIANE project. (http://www.ipd.uni-karlsruhe.de/DIANE/en)
3. Klein, M., König-Ries, B.: A process and a tool for creating service descriptions based on DAML-S. (http://www.ipd.uni-karlsruhe.de/DIANE/docs/KK03.pdf, submitted to 4th VLDB Workshop on Technologies for E-Services (TES'03))
4. Dumas, M., O'Sullivan, J., Heravizadeh, M., Edmond, D., Hofstede, A.: Towards a semantic framework for service description. In: 9th International Conference on Database Semantics, Hong-Kong, Kluwer Academic Publishers (2001)
5. Object Management Group: OMG IDL syntax and semantics. (http://www.omg.org/docs/formal/02-06-39.pdf)
6. Sun Microsystems: Java IDL. (http://java.sun.com/products/jdk/idl/)
7. Microsoft: Microsoft interface definition language. (http://msdn.microsoft.com/library/en-us/midl/midl/midl_start_page.asp)
8. World Wide Web Consortium: Web interface definition language. (http://www.w3.org/TR/NOTE-widl)
9. Sun Microsystems: Java RMI. (http://java.sun.com/products/jdk/rmi/)
10. Sun Microsystems: Jini. (http://www.jini.org/)
11. Sun Microsystems: Enterprise JavaBeans technology. (http://java.sun.com/products/ejb/)
12. World Wide Web Consortium: Web service description language (WSDL). (http://www.w3.org/TR/wsdl)
13. Hewlett Packard: HP web services platform. (http://www.hp.com/go/espeak)
14. ebXML: Collaboration protocol profile and agreement specification. (http://www.ebxml.org/specs/ebCCP.pdf)
15. Curbera, F., Weerawarana, S., Duftler, M.J.: Network accessible service specification language: An XML language for describing network accessible services. (http://www.cs.mu.oz.au/ eas/subjects/654/nassl.pdf)
16. Web-Ontology Working Group: Web ontology language - services (OWL-S). (http://www.daml.org/services/daml-s/0.9/)
17. Martin, D.L., Cheyer, A.J., Moran, D.B.: The open agent architecture: A framework for building distributed software systems. Applied Artificial Intelligence **13** (1999) 91–128
18. McIlraith, S., Son, T.C.: Adapting golog for composition of semantic web services. In: 8th International Conference on Knowledge Representation and Reasoning (KR2002). (2002) 482–493
19. Gurevich, Y.: Evolving algebras: An attempt to discover semantics. In Rozenberg, G., Salomaa, A., eds.: Current Trends in Theoretical Computer Science. World Scientific, River Edge, NJ (1993) 266–292
20. Sycara, K., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. Autonomous Agents and Multi-Agent Systems **5** (2002) 173–203
21. Klein, M., König-Ries, B., Obreiter, P.: Service rings – a semantical overlay for service discovery in ad hoc networks. In: The Sixth International Workshop on Network-Based Information Systems (NBIS2003), Workshop at DEXA 2003, Prague, Czech Republic. (2003)
22. World Wide Web Consortium: XML schema part 2: Datatypes. (http://www.w3.org/TR/xmlschema-2/)