Enhancing Semantic Web Services with Inheritance

Simon Ferndriger[1] Abraham Bernstein[1] Jin Song Dong[2]
Yuzhang Feng[2] Yuan-Fang Li[3*] Jane Hunter[3]

[1] Department of Informatics
University of Zurich
Zurich, Switzerland
ferndriger@gmail.com,bernstein@ifi.uzh.ch
[2] School of Computing
National University of Singapore, Singapore
{dongjs,fengyz}@comp.nus.edu.sg
[3] School of ITEE
University of Queensland
Brisbane, Australia
{liyf,jane}@itee.uq.edu.au

Abstract. Currently proposed Semantic Web Services technologies allow the creation of ontology-based semantic annotations of Web services so that software agents are able to discover, invoke, compose and monitor these services with a high degree of automation. The OWL Services (OWL-S) ontology is an upper ontology in OWL language, providing essential vocabularies to semantically describe Web services. Currently OWL-S services can only be developed independently; if one service is unavailable then finding a suitable alternative would require an expensive and difficult global search/match. It is desirable to have a new OWL-S construct that can systematically support substitution tracing as well as incremental development and reuse of services. Introducing inheritance relationship (IR) into OWL-S is a natural solution. However, OWL-S, as well as most of the other currently discussed formalisms for Semantic Web Services such as WSMO or SAWSDL, has yet to define a concrete and self-contained mechanism of establishing inheritance relationships among services, which we believe is very important for the automated annotation and discovery of Web services as well as human organization of services into a taxonomy-like structure. In this paper, we extend OWL-S with the ability to define and maintain inheritance relationships between services. Through the definition of an additional "inheritance profile", inheritance relationships can be stated and reasoned about. Two types of IRs are allowed to grant service developers the choice to respect the "contract" between services or not. The proposed inheritance framework has also been implemented and the prototype will be briefly evaluated as well.

1 Introduction

Current Web Services technology such as WSDL, UDDI, and SOAP provide the means to describe the "syntax" of the code running in a distributed fashion over the Internet.

* Author for correspondence: liyf@itee.uq.edu.au

They lack, however, the capabilities to describe the semantics of these code fragments, which is one of the major prerequisites for service recognition, service configuration and composition (i.e., realizing complex workflows and business logics with Web services), service comparison as well as automated negotiation.

To that end a number of languages such as OWL-S[4], SAWSDL[5], WSMO[6], and SWSF[7] have been proposed. Each of these languages allows connecting Web services with an ontology-based semantic description of what the service actually does. The OWL Services (OWL-S) ontology is an OWL ontology defining a set of essentia vocabularies to describe the "semantics" of Web services, defining its capabilities, requirements, internal structure and details about the interactions with the service. Other efforts provide similar vocabularies with different focus and coverage.

Based on the de-facto ontology language, OWL DL, OWL-S seems to be a promising candidate as an open standard. Currently OWL-S services can only be developed independently. Moreover, if one service is unavailable then finding a suitable alternative would require an expensive and difficult global search/match. It is desirable to have a new OWL-S construct that can support the systematic substitution tracing as well as the incremental development and reuse of services. Hence, in order for OWL-S to enjoy wider adoption, a more systematic, automated and effective mechanism of annotation and discovery of services is required.

The `owl:imports` construct of OWL can be seen as a rudimentary form of establishing links between OWL-S services to support easy service annotation. However, it does not provide the necessary flexibility since once a particular construct from a service ontology, say, a composite process in a service model, is imported, it can only be augmented by adding more triples describing it. Basically, the importing service cannot *revoke* any RDF statement already made in the imported ontology. Hence, only reusing constructs at very detailed level is possible for the importing approach, which we deem is neither desirable nor practical. An approach more flexible and powerful is needed.

Inspired by the object-oriented programming paradigm, we propose to extend OWL-S with service inheritance, which we believe improves the level of automation and effectiveness for carrying out the above tasks. So far, however, only the SWSF framework briefly discusses establishing connections between different Web services in order to reuse similar underlying elements and add additional relationship information. Furthermore, none of these standards defines a concrete and self-contained way of sharing specific elements among Web Services or a way of interpreting the relationship among these services.

1.1 Motivation

We believe adding inheritance relationships between services can help to automate and ease a number of tasks. In this subsection, we present some scenarios in which inheritance of services facilitates the completion of tasks.

[4] `http://www.daml.org/services/owl-s/`
[5] `http://www.w3.org/TR/sawsdl/`
[6] `http://www.wsmo.org/wsml/wrl/wrl.html`
[7] `http://www.daml.org/services/swsf/`

Semantic Service Annotation The number of Semantic Web services (SWS) needs to reach a critical mass in order for SWS to gain wider acceptance and adoption. Hence, the creation of semantic annotation of Web services is an important first task. Currently, with tool support, annotation of services are still mostly created from scratch. Inheritance mechanism can greatly speed up the annotation of Web services by selectively reusing components from existing services.
Service Discovery Automated Web service discovery is stated as a motivating task for OWL-S. Service discovery, however, depends heavily on (potentially large) service registries because there is yet no other way to discover those services otherwise.

An alternative to discovering relevant services without the need of a registry is to make use of inheritance relationships between services in order to find service substitutes more efficiently. Analogous to object-oriented concepts, when certain constraints are satisfied, a sub service may be used to substitute its super service for automated, dynamic service discovery and composition.

It may seem that existing language constructs such as `rdfs:subClassOf` can handle inheritance, by subclassing existing service annotations. However, as RDF Schema and OWL are based on monotonic logic, subclassing only represents a restricted form of inheritance.

Inspired by the MIT Process Handbook [1, 2] we believe that service ontologies are central to the organization of business knowledge. As shown by Malone and colleagues, process repositories that build on the inheritance of process properties can be effectively used to (1) invent new business processes, (2) systematically explore the design space of possible service alternatives through recombination [3], (3) design robust services through the advanced usage of exceptions, (4) support knowledge management about services by improving their management process, ability to handle conflicts, support for communicative genres, (5) as well as improve software design and generation by increasing the coordination alternatives between pieces of code and achieve the flexible execution of workflows [4].

Based on the above motivating tasks, we propose to extend OWL-S with Inheritance Relationships (IRs) between services for more automated annotation and discovery. We draw inspirations from the Semantic Web Services Framework (SWSF) and expand the brief discussion in SWSF on inheriting and overriding processes among services.

In this paper, we present an inheritance framework for OWL-S ontology. Two versions of Inheritance Relationships are supported: normal and strict inheritance for OWL-S in the form of additional, independent service profiles. The *normal inheritance* does not impose additional restrictions on the inheritance relationship in order to allow for more flexible reuse of existing service components. As normal inheritance inevitably allows the alteration of existing services, a form of default inheritance advocated by SWSL [5] is employed. The *strict inheritance*, by imposing certain restrictions on IOPEs of the inherited process, dictate that the "contracts" of processes of a super service must be maintained by the inheriting service. This guarantees a proper refinement relationship between the super service and the sub service. Hence, a strictly inheriting sub service can substitute its super service whenever the super service is invoked, whereas this substitutability is not guaranteed with normal inheritance. Moreover, it enables a sub service to be more easily discovered.

The rest of the paper is organized as follows. Section 2 briefly presents background knowledge about OWL-S and SWSF. Section 3 discusses the two forms of inheritance relationships in detail. In Section 4, we extend the well-known CongoBuy example from OWL-S specification to illustrate the benefits of IRs. Finally, Section 5 concludes the paper and discusses future work directions.

2 Background Knowledge

In this section, we introduce the background knowledge necessary for the discussion of the following sections.

2.1 OWL-S

The OWL-S ontology has been developed to enrich Web Services with semantics. The semantic markup of OWL-S enables the automated discovery, invocation, composition, interoperation and monitoring of Web services. This automation is achieved by providing a standard ontology (OWL-S) for declaring and describing Web Services.

Being an OWL ontology, OWL-S defines a set of essential vocabularies to describe the three components of a service: profile, model and grounding. A service can have several profiles and one service model. The service model, in turn, may have one or more service groundings. In summary, a service profile describes what the service does; the service model describes how the service works and the grounding provides a concrete specification of how the service can be accessed.

The `ServiceProfile` class provides a bridge between service requesters and service providers. The instances are mainly meant to advertise an existing service by describing it in a general way that can be understood both by humans and computer agents. It is also possible to use a service profile to advertise a needed service request.

OWL-S provides a subclass of `ServiceProfile`, `Profile`. This default class should include provider information, a functional description and host properties of the described service. It is possible to define other profile classes that specify the service characteristics more precisely.

The `ServiceModel` class uses the subclass `Process` to provide a process view on the service. This view can be thought of as a specification of the ways a client may interact with a service. The service model defines the inputs, outputs, preconditions and effects (IOPEs) and the control flow of composite processes.

One useful language construct in the service model is the definition of `Expression`, which is used to express preconditions and effects in the logic language of choice by the service developer. Basically, an expression is characterized by an expression language, such as SWRL [6], KIF [7], etc., and an expression body, containing the logic expression in that language.

The `ServiceGrounding` class provides a concrete specification of how the service can be accessed. Of main interest here are subjects like protocol, message formats, serialization, transport and addressing. The grounding can be thought of the concrete part of the Semantic Web service description, compared to the service profile and service model which both describe the service on an abstract level.

2.2 SWSF

The Semantic Web Services Framework (SWSF) [8] includes two major components, The Semantic Web Services Language (SWSL) [5] and the Semantic Web Services Ontology (SWSO) [9]. SWSL is a generic language, used in the SWSF framework to formally specify Web service concepts and descriptions. It includes two sublanguages: SWSL-FOL (based on first-order logic) and SWSL-Rules (based on logic programming). SWSO serves essentially the same purposes as OWL-S: providing semantic specifications of Web services; namely (similarly to OWL-S) a comparable service profile, model and grounding.

However, SWSO also has a number of significant differences from OWL-S.

- Higher expressivity: the SWSF service ontology (called FLOWS) is expressed in first-order logic. OWL-S, in contrast, is expressed in OWL-DL, a variant of description logic language $\mathcal{SHOIN}(D)$ [10].
- Enhanced process model: SWSF claims to provide an enhanced process model as compared to OWL-S. It is based on the Process Specification Language [11], hence, it provides Web Services specific process concepts that include not only inputs and outputs, but also messages and channels.
- Non-monotonic language: In addition to the service language, SWSO makes use of SWSL-Rules, a non-monotonic language based on the logic-programming paradigm which is meant to support the use of the service ontology in reasoning and execution environments.
- Interoperability: an important final distinction between OWL-S and FLOWS is with respect to the role they play. Whereas both endeavor to provide an ontology for Web services, FLOWS had the additional objective of acting as a focal point for interoperability, enabling other business process modeling languages to be expressed or related to FLOWS.

Molecules are a language construct in the Frames layer of the SWSL-Rules language. We will use molecules to present some of the inheritance-related concepts in later sections. Here, we give a brief overview of molecules. A molecules can be viewed as an atomic term: a constant, a variable or an function application. Of the seven forms of molecules, we present the two forms that will be used in this work: value molecules and boolean molecules. In this paper, the molecules will be presented in `teletext` font.

Value molecules are of the form `t[m -> v]` where `t`, `m` and `v` are all terms where `t` denotes an object, `m` denotes a function invocation in the scope of `t` and `v` denotes a value returned by the invocation. The molecule `t[m *-> v]` denotes that this method is inheritable.

Boolean molecules are of the form `t[m]` where `t` and `m` are both terms. A Boolean molecules can be interpreted as `t[m -> true]`, meaning that the property `m` of object `t` is true.

Complex molecules can be formed from other molecules by grouping and nesting. For example, the molecules `t[m -> v] and t[p]`, which describe the same object `t`, can be grouped together to form the complex molecule `t[m -> v and p]`. Similarly, `t[m -> v] or t[p]` can be grouped to form `t[m -> v or p]`.

2.3 Related Works

The concept of inheritance is not new. It has been an active research area in programming languages and software engineering for over decades. In particular, the works on behavioral subtyping [12, 13] of object-oriented languages and object-oriented specification languages are particularly related.

OWL-S defines an process hierarchy ontology[8] that describe a profile-based approach of creating service hierarchies. However, this approach, as the authors put it, "provides a useful means of constructing a 'yellow pages' style of service categorization". It does not support the extension/modification of services at the level of granularity presented in this paper.

3 IR Framework for OWL-S

Although inheritance has been widely used in computer science as a tool to encapsulate and manage program complexity and to improve code reuse and reliability, it has not been widely applied to the Web Services domain.

In this section, we present in detail our proposed inheritance relationships (IRs) framework for OWL-S services. The language constructs used to extend/modify inherited entities and conditions that must be satisfied by these constructs are presented.

We start this section with a discussion on the distinction between different types of inheritance relationships: normal vs strict inheritance and single vs multiple inheritance..

3.1 The Perspectives of Inheritance

Normal IR, closely related to default inheritance [14, 15], allows for flexible alteration of inherited service components. It primarily facilitates the easy annotation of Web services.

In complete inheritance, information that is used by more than one element has to be stored in a more general element. This means that no redundant information is allowed and information has to be inherited down the inheritance chain: the generalization must be complete. Therefore, inherited information can neither be altered nor arbitrarily extended.

On the contrary, default inheritance is defined such that elements get inherited by default which can be modified and extended afterwards. Hence, new features/functionalities are allowed to be added in default inheritance. In the Web environment, it is often the case that an inheriting service intends to extend the functionality of the inherited service. Compared to the OWL `imports` approach, default inheritance allows for the possibility of freely modifying an inherited entity. Furthermore, default inheritance has been shown to be easier to understand by non-specialists [16] making it more suitable for the wide variety of users on the Semantic Web. For those reasons, default inheritance is adopted for this approach.

[8] `http://www.daml.org/services/owl-s/1.1/ProfileHierarchy.owl`

Strict IR aims at enabling more automated and accurate service discovery by following the inheritance chain between services. Seen as a form of normal IR, strict IR imposes certain restrictions such that a strictly inheriting service can automatically be used as a faithful substitute for the inherited service.

The faithful substitute is achieved by following the principle of operational refinement [17] on the IOPEs of the inherited processes. Briefly, let the IOPEs of an OWL-S service process SP_{sb} and its ancestor process SP_{sp} be I, O, P, E and I', O', P', E', respectively. In order to establish a strict IR between S_{sb} and S_{sp}, the following conditions must hold.

$P' \Rightarrow P \qquad E \Rightarrow E'$

e.g., the preconditions P of the inheriting service SP_{sb} must be weaker than that of the inherited service, and vice versa for the effects. These follow from the well-established data refinement principles and covariance.

Besides preconditions and effects, the inputs and output must also satisfy similar constraints: (a) the number and all names of input/output parameters must match (up to permutation) and (b) for each matching input parameter in I and I', the type of parameter of the inheriting service must be a subtype of that of the inheriting service and (c) vice versa for the output parameters.

Single inheritance, from an orthogonal point of view, only allows an element to inherit from a single more general class.

Multiple inheritance has the advantage over single inheritance of providing the ability to inherit functionalities from several super services. It does, however, add additional complexity which might lead to inconsistencies such as catcalls [18].

Like normal inheritance, multiple inheritance enable service developers to reuse multiple services conveniently. Therefore, it is also incorporated. Hence, in our framework, default inheritance and multiple inheritance are allowed.

3.2 IR Syntax Extension

A number of OWL classes and properties are used to construc IRs and connect services to them. IRs are modeled in additional, independent service inheritance profiles, which are modeled as a subclass of the OWL-S class `ServiceProfile`. The inheritance can be modeled in two directions, meaning that a service can point to its super services, as well as its sub services. The `InheritanceProfile` is connected to the specific super/sub service through the *abstract* class `Relationship`, which is defined to be the disjoint union of classes `SuperService` and `SubService`, and the property `contains`. Definition 1 below defines inheritance profile and how it is linked to OWL-S services. The OWL code fragment is presented in the familiar "DL syntax" and in *math* font.

$$\begin{array}{ll} \mathit{InheritanceProfile} \sqsubseteq \mathit{ServiceProfile} & \geq 1\, \mathit{contains} \sqsubseteq \mathit{InheritanceProfile} \\ \mathit{Relationship} = & \top \sqsubseteq \forall\, \mathit{contains}.\mathit{Relationship} \\ \quad \mathit{SuperService} \sqcup \mathit{SubService} & \mathit{SuperService} \sqcap \mathit{SubService} = \bot \end{array} \tag{1}$$

Processes in service models are inheritable. Note that since the grounding of a service specifies the concrete physical Web service, groundings are not inheritable as an inherited service is a new service.

Note that in an inheritance profile, a service can refer to either its super service or its sub service. A service can have multiple super or sub services. A particularly interesting scenario arises when an inheritance relationship is stated in both the super service and sub service. In this case, the inheriting service may be interpreted as "endorsed" by the super service. Hence, if the super service is trusted, the sub service can also be trusted.

We distinguish two types of IRs in OWL-S: normal and strict (details are given in the subsection below). As introduced previously, the two types of inheritance tackles different problems an service developers have the freedom to choose the appropriate form. An OWL object property `fromType` and an enumerated class `Type` (with two instances `normal` and `strict`) are used to state whether a particular IR is normal or strict. An IR relationship has exactly one inheritance type[9].

The modification of default inheritance is modeled by an OWL property `specifiedBy`, with `SuperService` as its domain and the "abstract" class `Specification` as its range. The class `Specification` is a class with three disjoint subclasses: `Customization`, `Extension` and `Manipulation`. These three different types of modification are used to modify an inherited service and will be presented in detail later in this section.

As stated in Section 1, default inheritance alteration can be specified by SWSL-Rules language. In modeling IR, we define an OWL class `SWSL-Expression` and a property `externallySpecifiedBy` to link an normal IR to the SWSL-Rules expression that modifies it. An instance `SWSL` of the class `LogicalLanguage` (defined as part of the OWL-S framework) is also defined to represent the logic language SWSL.

Therefore, a super service can be further modified by at most one `Specification` or at most one SWSL expression, as given in Definition 2 below.

$$\textit{SuperService} \sqsubseteq (\leq 1\, \textit{specifiedBy} \sqcup \leq 1\, \textit{externallySpecifiedBy}) \tag{2}$$

Finally, in order to link the current service to its super/sub service. An OWL object property `hasSource` is defined, with `Relationship` as its domain and `Service` as its range.

Modification of Inheritance In this subsection, we describe the language extensions used to modify the inheritance relationships between OWL-S services.

In Table 1 below, we briefly introduce the main differences among of the three types of modification of inherited services, which will be presented in more detail in the following subsections.

The language constructs in inheritance modification can be divided into three scenarios: customization, extension and manipulation, based on their intended usage. For better readability, the scenarios are presented in SWSL molecule syntax [5]. The same modeling can also be represented in OWL, which is more verbose.

Service customization allows one to choose whether to inherit service model; rename and replace an inherited entities (processes and parameters).

[9] For brevity reasons, the formal OWL definitions are not shown when not necessary. Full details of the modeling can be found online at `http://www.fo-ss.ch/simon/DiplomaThesis/InheritanceProfile/InheritanceProfile.owl`.

Table 1. Main differences between service customization, extension & manipulation

Type	IR type	Modification
Customization	strict	process & IDs
Extension	normal	process
Manipulation	normal & strict	IOPEs

```
Inherit[AdoptServiceModel(PID_INHERITED)]                                    (3)
Renaming[ID_INHERITED *-> ID_REPLACEMENT]
ProcessReplacement[PID_INHERITED *-> PID_REPLACEMENT]
```

where $\texttt{PID}_{\texttt{INHERITED}}$ stands for the inherited process ID, which is to be replaced. $\texttt{PID}_{\textit{REPLACEMENT}}$ represents the replacing process. The SWSL method `AdoptService Model` enables one to use one of the inherited processes from the super service as the new service model.

In the `Inherit` molecule, the term `Processes` must always be included. Therefore, by default, a copy of the ontology which contains the service model of the super service gets integrated into the service ontology of the sub service. Note that since an OWL-S service can only have one service model, in case of multiple inheritance, only the service model of one of the super services can be inherited.

For the molecules `Renaming` and `ProcessReplacement` it must be ensured that the new IDs do not conflict with existing ones. Moreover, in `ProcessReplacement`, the IOPEs of the replacement process must match those of the inherited process. In other words, the refinement relationship must be maintained.

Service extension allows one to extend an inherited process model by inserting into, detaching from or deleting an inherited process or the perform of the process.

```
ProcessInsertion[{after/before} PPID_INHERITED *-> CCID_NEW]                 (4)
ProcessDeletion[PPID_INHERITED]
```

where the expression $\{\texttt{after/before}\}$ means that the new process can be inserted either before or after the process perform. This allows on to directly use a process perform which is connected with a process, or one can wrap the process perform into another control construct (e.g. if-then-else or sequence, etc). The Boolean molecule `ProcessDeletion` models the fact that a process can be deleted from the inherited model.

Service manipulation allows one to modify the preconditions and effects of an inherited process. Moreover, in normal inheritance mode, service modification also allows one to add/remove the input/output of the inherited process.

```
ExpressionReplacement[{                                                      (5)
    replaceCondition(CID_1) *-> CID_2 or
    replaceResult(CID_1) *-> CID_2
}]
AddInputsAndOutput[addIO(PID, OID) ->* NID]
DeleteInputsAndOutput[deleteIO(PID, OID) ->* NID]
```

where $\texttt{CID}_1$ represents the ID of the replaced precondition/result and $\texttt{CID}_2$ represents the ID of the new one. `NID` and `OID` represent the RDF ID of the new and old

input (or output) parameter, respectively. `PID` represents the ID of the process where the parameter is added/deleted.

When used in strict inheritance, the replacing precondition/effect must still satisfy the refinement relationship between the inherited and the current process, e.g., the replacing precondition must be weaker than the inherited precondition and vice versa for effects.

3.3 Satisfaction Conditions of IR

In this subsection, we present some conditions that the modifications of default inheritance presented above must satisfy. These conditions guarantee that, for example, the modifications allow proper process flow in case of strict inheritance. The following conditions are presented in first-order logic syntax with some notations/elements taken from the OWL abstract syntax and semantics [19].

Generally, these conditions need to be checked by software agents making use of the IRs. How the conditions are checked may be application-specific.

Service customization. Since an OWL-S service can only have one service model. in case of multiple-inheritance (i.e., multiple IRs), the service model can only be adopted once by a sub service, respectively it can only be inherited from one of all of its super services. This condition is formally captured in the following first-order predicate. Note that *O*, *EC*, *ER* and *LV* in the conditions below are entities of the abstract interpretation defined in OWL semantics [19].

$$
\begin{aligned}
&\forall\, IHP, SS_1, SS_2, SP_1, SP_2, SM_1, SM_2 : O \bullet \\
&\quad SS_1 \in EC(SuperService) \wedge SS_2 \in EC(SuperService) \wedge \\
&\quad SP_1 \in EC(Inherit) \wedge SP_2 \in EC(Inherit) \wedge SM_1 \in EC(ServiceModel) \wedge \\
&\quad IHP \in EC(InheritanceProfile) \wedge SM_2 \in EC(ServiceModel) \wedge \\
&\quad \langle IHP, SS_1 \rangle \in ER(contains) \wedge \langle SS_1, SP_1 \rangle \in ER(specifiedBy) \wedge \\
&\quad \langle IHP, SS_2 \rangle \in ER(contains) \wedge \langle SS_2, SP_2 \rangle \in ER(specifiedBy) \wedge \\
&\quad \langle SP_1, SM_1 \rangle \in ER(adoptServiceModel) \wedge \langle SP_2, SM_2 \rangle \in ER(adoptServiceModel) \\
&\qquad \Rightarrow \\
&\quad SS_1 = SS_2 \wedge SM_1 = SM_2
\end{aligned}
\tag{6}
$$

Formally, Definition 6 above specifies that for an arbitrary inheritance profile *IHP* (for an OWL-S service), if it *contains* two super services SS_1 and SS_2, and adopts the service model of each of these two services, then the two services are actually one service ($SS_1 = SS_2$) and the two service models are one model ($SM_1 = SM_2$) as well.

Similarly for renaming of IDs, it must be ensured the the original ID must be present in the inherited service and the new ID must not conflict with existing ones.

$$
\begin{aligned}
&\forall\, SS, SP : O;\ XID : LV \bullet \exists\, X : O;\ OID : LV \bullet \\
&\quad SS \in EC(SuperService) \wedge SP \in EC(Renaming) \wedge \langle SS, SP \rangle \in ER(specifiedBy) \wedge \\
&\quad \langle SP, XID \rangle \in ER(oldID) \wedge (X, OID) \in ER(ID) \wedge X \in SS \\
&\qquad \Rightarrow \\
&\quad XID = OID
\end{aligned}
\tag{7}
$$

The slight "abuse" of syntax in predicate $X \in SS$ above means that X is bound in SS. The above condition guarantees that the replaced ID is always present in the inherited service. The condition for new ID can be similarly defined.

The conditions for a process replacement are more complicated. First of all, the input and output IDs of the process replacement must match (Definitions 8 below specifies that for the inputs) and their types must be compatible. When input types of the replacement process are OWL classes, the input types must either be from the same OWL class or from an OWL sub class of the original ones, similarly for data types. It is specified in the second condition below.

Note that the formula of syntax $\{decl \mid pred \bullet proj\}$ is a set comprehension expression, meaning that for variables declared in *decl* part, the set contains elements specified in *proj* that satisfy the conditions specified in *pred*.

$$
\begin{array}{l}
\forall PR, OP, RP : O;\ OIDs, RIDs : \mathbb{P}\, LV \bullet \\
\quad PR \in EC(ProcessReplacement) \wedge \\
\quad \langle PR, OP \rangle \in ER(replaceProcess) \wedge \\
\quad OIDs = \{OID : LV, OI : O \mid \\
\qquad \langle OP, OI \rangle \in ER(hasInput) \wedge \\
\qquad \langle OI, OID \rangle \in ER(ID) \bullet OID\} \wedge \\
\quad \langle PR, RP \rangle \in ER(withProcess) \wedge \\
\quad RIDs = \{RID : LV, RI : O \mid \\
\qquad \langle RP, RI \rangle \in ER(hasInput) \wedge \\
\qquad \langle RI, RID \rangle \in ER(ID) \bullet RID\} \wedge \\
\quad \Rightarrow \\
\quad OIDs = RIDs
\end{array}
\qquad
\begin{array}{l}
\forall PR, OP, RP, RI : O; \\
\quad RT : V_r;\ RID : LV \bullet \\
\qquad \exists OI : O;\ OT : V_o \bullet \\
\qquad PR \in EC(ProcessReplacement) \wedge \\
\qquad \langle PR, OP \rangle \in ER(replaceProcess) \wedge \\
\qquad \langle OP, OI \rangle \in ER(hasInput) \wedge \\
\qquad \langle PR, RP \rangle \in ER(withProcess) \wedge \\
\qquad \langle RP, RI \rangle \in ER(hasInput) \wedge \\
\qquad \langle OI, OID \rangle \in ER(ID) \wedge \langle RI, OID \rangle \in ER(ID) \wedge \\
\qquad OI \in EC(V_o) \wedge RI \in EC(V_r) \\
\qquad\quad \Rightarrow \\
\qquad EC(V_r) \subseteq EC(V_o)
\end{array}
\tag{8}
$$

Secondly, the preconditions and effects of the two processes must comply with the refinement concept, e.g., the preconditions of the modified process must be weaker than those of the original process. In case of multiple preconditions (each for a different scenario), their conjunction is taken into consideration.

$$
\begin{array}{l}
\forall PR, OP, RP : O;\ OPCs, RPCs : \mathbb{B} \bullet \\
\quad PR \in EC(ProcessReplacement) \wedge \langle PR, OP \rangle \in ER(replaceProcess) \wedge \\
\quad OPCs = \bigcap (OPC : \mathbb{B} \bullet \langle OP, OPC \rangle \in ER(hasPrecondition)) \wedge \\
\quad \langle PR, RP \rangle \in ER(withProcess) \wedge \\
\quad RPCs = \bigcap (RPC : \mathbb{B} \bullet \langle RP, RPC \rangle \in ER(hasPrecondition)) \\
\qquad \Rightarrow \\
\quad RPCs \Rightarrow OPCs
\end{array}
\tag{9}
$$

Note that the symbols $\bigcap$ and $\bigcup$ represent distributed set intersection/union, respectively. For brevity reasons, those conditions regarding outputs and effects are omitted but they can be similarly defined.

Service manipulation. When service manipulation is used in strict mode, only the expression replacement statement can be made as the rest two (adding and removing inputs/outputs) would violate that relationship. For the first statement, it needs to be ensured that the altered precondition is logically weaker and the effect is stronger. The following two conditions model the case for preconditions and effects, respectively.

$$\forall EP, RP, CP : O;\ OPC, RPC : \mathbb{B} \bullet \quad \forall EP, RP, CP : O;\ OPR, RPR : \mathbb{B} \bullet \tag{10}$$
$$EP \in EC(ExpressionReplacement) \land \quad EP \in EC(ExpressionReplacement) \land$$
$$\langle EP, OPC \rangle \in ER(replaceCondition) \land \quad \langle EP, OPR \rangle \in ER(replaceResult) \land$$
$$\langle EP, RPC \rangle \in ER(withCondition) \quad \langle EP, RPR \rangle \in ER(withResult)$$
$$\Rightarrow \quad \Rightarrow$$
$$RPC \Rightarrow OPC \quad OPR \Rightarrow RPR$$

3.4 Some Discussion on IR

During service discovery, the relationship may need to be interpreted and validated to ensure that it is valid. When there is a chain of inheritance, the validation/interpretation should be performed top down in order for inheritance information to propagate properly.

One interesting scenario may arise when a super service is modified, after an IR is established between it and a sub service. In this situation, it might be necessary in certain situations to revalidate the inheritance relationship.

Another reason to revalidate an IR is to benefit from possible side effects of such a revalidation. In case a super service changes not its service groundings, but its process composition, such a change would not affect the corresponding sub service as long as the IR stays valid.

For example, the super service could have a "Search Flight" process, which changes from being atomic to being composite in order to make it more efficient. Since this change happens in the service model which gets copied to the sub service during the validation of the IR, the service model of the sub service needs first to be updated by revalidating the IR in order to adopt this change.

Theoretically, service substitutes can not only be found via strict IR, but also via ordinary on-the-fly reasoning in a service registry using the refinement relationship as it is used for defining strict IRs. Without the IR relationship, however, this reasoning is likely to be very expensive in time, since every service has to be considered as a candidate substitute. Hence, IR helps to reduce service discovery time by providing guided exploration of service space and hence eliminating most necessary comparison.

4 The Benefits of IR – A Case Study

We have developed a prototype service repository[10] that implements the inheritance relationships framework presented in this paper. Through a Web interface, the prototype has the following four main functionalities: service annotation creation, service visualization, service discovery and inheritance validation.

In this section, we present one example in the service repository, the Congo book store web service from the OWL-S specification extended with inheritance. Congo is a fictitious online bookstore that uses OWL-S ontologies to semantically markup their services. We use both normal and strict IR in service annotation creation scenarios, demonstrating their benefits and differences.

[10] The prototype is accessible at `http://www.fo-ss.ch/simon/DiplomaThesis/IR_prototype/`.

4.1 Normal inheritance

FullCongoBuy is an example service published with the OWL-S specification. It provides a complete book buying service for physical books. In this example, we extend it with the capability of buying digital books also.

Given the existing FullCongoBuy service, it would be convenient to benefit from the work already done when creating the other book selling service E-BookBuy instead of starting from scratch. Without inheritance, the modifications needed to create E-BookBuy from FullCongoBuy is not possible

The proposed IR, however, makes it possible to reuse the service model of FullCongoBuy within the context of service customization and extension. The IR allows not only the reuse but also the necessary altering, i.e. deletion and replacement of inherited properties. More concretely, the new service annotation E-BookBuy can be created by inheriting the service model from FullCongoBuy (11), replacing the process LocateBook with a new one (12), deleting the process SpecifyDeliveryDetails-Perform (13) while adding SpecifyDownloadDetailsPerform as an alternative, adding a new process ProvideDownloadOptions (14), making a new service profile and creating the grounding for the new processes. For better readability, the main process can be renamed (15).

```
Inherit[AdoptServiceModel(FullCongoBuy), Processes].                (11)
ProcessesReplacement[LocateBook *-> Locate_eBook].                  (12)
ProcessDeletion[SpecifyDeliveryDetailsPerform].                     (13)
ProcessInsertion[                                                   (14)
    after(BuySequence) *-> SpecifyDownloadDetailsPerform,
    after(SpecifyDownloadDetails) *->
        ProvideDownloadOptionsPerform ].
Renaming[ FullCongoBuy *-> Full_eBookBuy ].                         (15)
```

4.2 Strict Inheritance

ExpressCongoBuy is an example service published with the OWL-S specification. It provides a one-step book buying service for the Congo shop with a standard delivery setting. In real life, however, there might be different delivery settings. Since a concrete delivery is not yet defined in the example, this use case defines a one-day delivery for ExpressCongoBuy and creates a new service annotation EconomyCongoBuy with a slower three-day delivery.

Given the existing ExpressCongoBuy service, it would be convenient to benefit from the work already done when creating the other book selling service EconomyCongoBuy instead of starting from scratch. Without inheritance, there is no way to benefit from existing atomic services in creating a new one other than using cut and paste.

The proposed IR, however, makes it possible to reuse the service model of ExpressCongoBuy within the context of service customization and manipulation. More concretely, the new service annotation EconomyCongoBuy can be created by inheriting the service model from ExpressCongoBuy, replacing the positive result and adding a new service profile and grounding. The necessary statements for this strict IR are described in SWSL below.

```
Inherit[AdoptServiceModel(ExpressCongoBuy), Processes ].
Renaming[ExpressCongoBuy *-> EconomyCongoBuy ].
ExpressionReplacement[
    Effect(ExpressCongoBuy, ExpressCongoOrderShippedEffect) *->
    Effect(EconomyCongoBuy, EconomyCongoOrderShippedEffect)
].
```

The benefits of using IRs in service creation can be summarized as follows.

- **Efficient service annotation creation**: First, the reuse of information facilitates the creation of the service annotations since the service model and groundings of existing services can be largely reused. Therefore, using IR can improve the efficiency of the creation of similar annotations. When normal IR is used, the process flow has to be taken care of by the service creator, and therefore, the creator has to be familiar with the original service model.
- **Additional relationship information**. The explicit statement of the strict IR provides additional information about the two services, which may be used later on to facilitate more smooth service discovery and substitute.

5 Conclusion

Semantic Web Services languages aim at providing semantic markups for Web Services description in order to facilitate automated service discovery, composition, monitoring and composition.

The OWL-S ontology provides a set of semantic service descriptions that describes a service's functionality, internal structure and interfacing information for software agents to automate the above task. However, the current OWL-S specification does not specify a systematic way of creating and discovering services, limiting its wider adoption.

In this paper, we attempt to tackle the above problem by proposing a (default) inheritance relationship (IR) between OWL-S services. The IR is modeled in inheritance profiles and a set of language constructs are provided to link a service to its super/sub services. In addition, two modes of IRs, normal and strict, and their respective applications are presented and compared.

For service annotation creation, this approach provides service customization, extension and manipulation for sharing and modifying specific elements inherited from super services. This ability is expected to substantially reduce the amount of work necessary for creating and maintaining services. For service discovery, the approach provides a solution to find service substitutes for the developed strict IRs, based on the concept of refinement. These substitutes increase the choice of a service user or the availability of services as a whole.

Additionally, the proposed IRs allow a service to point to both super and sub services. The benefits are twofold. Firstly, if a particular IR is specified in both the super and sub services, the inheritance relationship between the two can be seen as "endorsed" by the services. Therefore, a stronger sense of trust can be established. Secondly, when used extensively, the IRs may connect a potentially large amount of services and thereby

build a strong service graph without the need of a central registry. This facilitates the distributed development and discovery of services.

The well-known frame problem [20] identified by Borgida, Mylopoulos and Reiter applies to most pre/post-condition style object-oriented and procedure specification languages. The problem is concerned about unwanted effects of a precedure/operation resulted from under specification of pre/post conditions. Essentially, the problem is caused by the inability of a specification language to express that an operation changes "only" those things that it intends to, and nothing else. This problem is particularly serious for specification languages with inheritance, where a sub class is usually constructed by conjoining specifications of super classes with additional predicates, where conjoining predicates may result in inconsistent pre/post-conditions.

As our work presented in this paper adds inheritance to OWL-S services and processes, it inherently has the above problem. It is an important future research task to investigate the impact and possible solutions of frame problems on both normal and strict IRs.

In Section 3 we discussed the conditions that the various types of modifications of default inheritance must satisfy. The validation of these conditions may be a computationally intensive process. In practical environments, a service may be modified after it is inherited by some other process. In this case, the conditions may need to be re-validated to ensure that they still hold. One future work direction would be to investigate under which circumstances these conditions need to be re-validated.

Another future work direction is to further develop the prototype into a more robust and usable form. We are currently investigating the possibility of developing a Protégé [21] plugin based on the current prototype. This plugin could directly communicate with the existing OWL-S plugin and would, therefore, be better accessible for future Web services developers.

Given that the OWL-S service ontology serves similar purposes as other languages such as SWSO, WSMO and WSDL-S, the investigation of the transfer of the proposed inheritance relationships is an interesting and important direction to pursue.

References

1. Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborn, C.S., Bernstein, A., Herman, G., Klein, M., O'Donnell, E.: Tools for Inventing Organizations: Toward a Handbook of Organizational Processes. Manage. Sci. **45**(3) (1999) 425–443
2. Malone, T.W., Crowston, K., Herman, G.A.: Organizing Business Knowledge: The MIT Process Handbook. MIT Press, Cambridge, MA, USA (2003)
3. Bernstein, A., Klein, M., Malone, T.W.: The process recombinator: a tool for generating new business process ideas. In: ICIS '99: Proceeding of the 20th international conference on Information Systems, Atlanta, GA, USA, Association for Information Systems (1999) 178–192
4. Bernstein, A.: How can cooperative work tools support dynamic group process? bridging the specificity frontier. In: CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work, New York, NY, USA, ACM (2000) 279–288

5. Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., Tabet, S.: Semantic Web Services Language (SWSL). http://www.daml.org/services/swsf/1.0/swsl/ (2005)
6. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/ (May 2004)
7. Ginsberg, M.L.: Knowledge Interchange Format – draft proposed American National Standard (dpANS). Technical Report 2/98-004, Stanford University (1998)
8. Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruniger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., Tabet, S.: Semantic Web Services Framework (SWSF). Technical report, Semantic Web Services Initiative (SWSI) (April 2005)
9. Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., McIlraith, S., McGuinness, D., Su, J., Tabet, S.: Semantic Web Services Ontology (SWSO). Technical report, Semantic Web Services Initiative (May 2005)
10. Horrocks, I., Patel-Schneider, P.F.: Reducing OWL entailment to description logic satisfiability. In Fensel, D., Sycara, K., Mylopoulos, J., eds.: Proc. of the 2003 International Semantic Web Conference (ISWC 2003). Number 2870 in Lecture Notes in Computer Science, Springer (2003) 17–29
11. Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J., Lee, J.: The Process Specification Language (PSL): Overview and Version 1.0 Specification. Technical Report NISTIR 6459, National Institute of Standards and Technology (2000)
12. America, P.: Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J.W., de Roever, W.P., Rozenberg, G., eds.: Foundations of Object-Oriented Languages. Volume 489 of Lect. Notes in Comput. Sci., Springer-Verlag (1991) 60–90
13. Findler, R.B., Latendresse, M., Felleisen, M.: Behavioral contracts and behavioral subtyping. In: ACM SIGSOFT '01: Proceedings of the 9th Symposium on the Foundations of Software Engineering (FSE'01). (2001)
14. Briscoe, T., Copestake, A., de Paiva, V., eds.: Inheritance, Defaults and the Lexicon. Cambridge University Press, New York, NY, USA (1993)
15. Daelemans, W., De Smedt, K.: Default Inheritance in an Object-oriented Representation of Linguistic Categories. International Journal Human-Computer Studies **41** (1994) 149–177
16. MacLean, A., Carter, K., Lövstrand, L., Moran, T.: User-tailorable systems: pressing the issues with buttons. In: CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, ACM (1990) 175–182
17. Morgan, C.: Programming from Specifications. International Series in Computer Science. Prentice-Hall (1990)
18. Meyer, B.: Static Typing. In: OOPSLA '95: Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum), New York, NY, USA, ACM (1995) 20–29
19. P. F. Patel-Schneider and P. Hayes and I. Horrocks (editors): OWL Web Ontology Semantics and Abstract Syntax. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/ (2004)
20. Borgida, A., Mylopoulos, J., Reiter, R.: On the Frame Problem in Procedure Specification. IEEE Transactions on Software Engineering (1995)
21. Gennari, J., Musen, M.A., Fergerson, R.W., Grosso, W.E., Crubézy, M., Eriksson, H., Noy, N.F., Tu, S.W.: The evolution of protégé: An environment for knowledge-based systems development. Technical Report SMI-2002-0943, Stanford Medical Informatics, Stanford University (2002)