Managing the Evolution of Simple and Complex Mappings between Loosely-Coupled Systems

Yu Deng, Harumi Kuno, Kevin Smathers
Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto
HPL-2004-68
June 11, 2004*

E-mail: yuzi@cs.umd.edu, {firstname.lastname}@hp.com

Semantic standards and technologies can facilitate the management of the dependency relationships between related components. Applications at different layers must map shared information both structurally (e.g., identifying which fields map to each other in different data schemas) as well as semantically (e.g., specifying the derivation relationship between a field in one schema and one or more fields from a potentially different schema). Believing that the efficient maintenance and evolution of such mappings is as important as creating the initial mappings in the first place, we present here methodologies for maintaining both simple and semantically complex mappings that exploit the rich constructs that Semantic Web technologies offer for modelling relationships. We include a description of a reference implementation that focuses on the domain of resource managing and provisioning.

* Internal Accession Date Only

Approved for External Publication

Managing the Evolution of Simple and Complex Mappings between Loosely-Coupled Systems*

Yu Deng, Harumi Kuno, and Kevin Smathers

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
yuzi@cs.umd.edu, [firstname.lastname]@hp.com

Abstract. Semantic standards and technologies can facilitate the management of the dependency relationships between related components. Applications at different layers must map shared information both structurally (e.g., identifying which fields map to each other in different data schemas) as well as semantically (e.g., specifying the derivation relationship between a field in one schema and one or more fields from a potentially different schema). Believing that the efficient maintenance and evolution of such mappings is as important as creating the initial mappings in the first place, we present here methodologies for maintaining both simple and semantically complex mappings that exploit the rich constructs that Semantic Web technologies offer for modelling relationships. We include a description of a reference implementation that focuses on the domain of resource managing and provisioning.

1 Introduction

The Semantic Grid brings to mind the visions for dynamically composed Web services that emerged, then faded, a few years ago. Just as Web standards and protocols such as HTML and HTTP create a single image from information resources that span the world, Grid technology, standards and protocols strive to make a disparate variety of distributed computing and data resources accessible through a unified virtual interface. Many researchers are currently exploring the combination of grid computing and Web services into a service-oriented architecture. Insofar as grid middleware strives to support the dynamic combination of distributed resources, this aspect of the grid vision recalls the inter-enterprise dynamic service composition that were prevalent in the early days of Web services, when UDDI was being touted as a means for services fronted by autonomous companies to discover and interact with each other.

The Web service vision eventually subsided to address the immediately practical problem of intra-enterprise application integration. On the one hand, this might seem a dilution of the Web service vision. However, another perspective is that this focus reflects the fact that as companies have grown in the size and

* This technical report is an extended version of [7].

complexity, enterprise systems became more loosely-coupled and integration thus became less simple. Web service standards and technologies simplify the design and use of integration middleware, and thus gained acceptance.

We think this practical need for integration presents a key opening for Semantic Grid. The Semantic Grid vision applies Semantic Web technologies to all aspects of grid computing, from the infrastructure fabric to applications [5]. As noted in [11], large scale resource integration is the essence of the grid. The Global Grid Forum (GGF)'s Open Grid Services Infrastructure (OGSI) [10] defines mechanisms for creating, managing, and exchanging information among grid services. Integrating and correlating relevant information thus presents a significant challenge. Resources from disparate systems must be allocated, assigned, composed and managed uniformly. Constraints, policies and SLAs must be interpreted in the context of data, messages, and events that originate from diverse information domains. In order for a management system to locate, isolate, and diagnose problems in a distributed environment, business virtualization software must correlate performance details from the service layer with fault and bottleneck data collected at the application layer as well as with configuration information and CPU, memory, and I/O metrics from the resource management layer [23]. The various systems must support lifecycle changes as applications and services evolve and are decommissioned.

Ontology languages (e.g., OWL) offer standard mechanisms for structuring knowledge about entities in different domains in terms of classes, relationships, and inference rules that operate upon those classes and relationships. A number of researchers have proposed mechanisms for discovering and expressing mappings between individual terms in different ontologies [8, 13]. However, they do not capture algebraic relationships or dependencies between the properties expressed by the ontologies, nor do they (in general) address efficient mechanisms for maintaining such mappings over the lifetime of a system.

In this paper, we address how to manage the evolution of both simple and complex mappings between ontologies representing loosely-coupled domains. We exploit unique features of OWL in order to make three main contributions. First, we extend OWL with constructs to create and maintain virtual properties– properties whose values are derived functionally instead of stored. These virtual properties can be used to express complex mappings between ontology terms. Second, we provide a methodology for evolving ontologies so as to maintain existing mappings. Third, we combine our evolution methodology and constructs for managing virtual properties, and show how to identify virtual properties that are affected by a change to an ontology.

2 Semantic Web Technologies

The Semantic Web initiative provides standards for associating machine accessible semantics with metadata [16, 6, 26]. The Resource Description Framework (RDF) defines a model for describing relationships among resources in terms of

uniquely identified properties (attributes) and values. The fundamental elements of RDF are:

- *Resource*: A resource is anything identified by a URI (plus optional anchor ids).
- *Literal*: A literal is an atomic value (e.g., integer or string).
- *Property*: A property is a resource that represents a specific aspect, characteristic, attribute, or relation used to describe resources. The set of properties is a subset of the set of resources.
- *Statement*: A statement is an ordered triple that associates a specific resource with a named property and a value of that property for that resource. The components of the statement are called, respectively, the subject, the predicate, and the object. The object of a statement can be either a resource or a literal. RDF has a mechanism, reification, for transforming a statement into a resource with a URI.

In object-oriented terms, we might consider RDF resources to be analogous to objects, RDF properties to represent attributes, and RDF Statements to express the attribute values of objects. A key difference between the two communities is that unlike OO systems, which use the concept of a type hierarchy to constrain the properties that an object may possess, RDF permits resources to be associated with arbitrary properties; statements associating a resource with new properties and values may be added to an RDF fact base at any time. For a brief discussion further comparing the RDF type system to an object-oriented model, the reader is referred to Appendix B.

RDF is designed for the representation and processing of metadata about information (resources) that are accessible through the Web, and is thus characterized by a property-centric, flexible and dynamic data model. In the RDF model, resources can gain and acquire new properties without any constraints (at any time, regardless of the type of the resource/property). This flexibility makes RDF an attractive technology for the specification and exchange of arbitrary metadata, because resource descriptions are "grounded" without being bound by fixed schemas.

Standards such as RDF Schema and OWL extend the fundamental RDF concepts of resource, literal, property, and statement with properties that specify formal, logic-based semantics, supporting the deduction of implicit knowledge (statements) from explicitly represented knowledge [1]. RDFS introduces properties that represent the relationships between RDFS classes and properties.

Such constructs enable extremely complex relationships to be specified using a comparably small number of "rules." Properties express relationships between resources, and, especially when combined with the constraints and rules defined by RDF Schema and OWL, provide a flexible and powerful model for representing resource composition. Furthermore, because properties are themselves resources, we can also use RDF-based technologies to formally express the relationships between properties–even from different domains/namespaces. RDF and related standards thus offer flexible and dynamically extensible support for complex, evolving, mappings between domains.

Virtual property. OWL does not provide native support for property chaining, arithmetic, view definitions, or procedural attachments [26]. However, these features are required in order to represent complex relationships between entities in the enterprise. OWL itself is neither a constraint nor a policy language, nor does it support explicit aggregation relationships.

In a loosely-coupled system, such as utility data center[22], the instance data schemas may be different from the abstract resource schemas, as illustrated in Figure 1. We believe that a *virtual property* construct –a property whose value is functionally derived–is required in order to represent the semantically complex mappings between resources spanning different schemas, and thus introduce this concept here. The top half of Figure 1 shows a resource schema (RS) in a utility grid while the bottom half shows a instance schema (IS) that RS maps to.

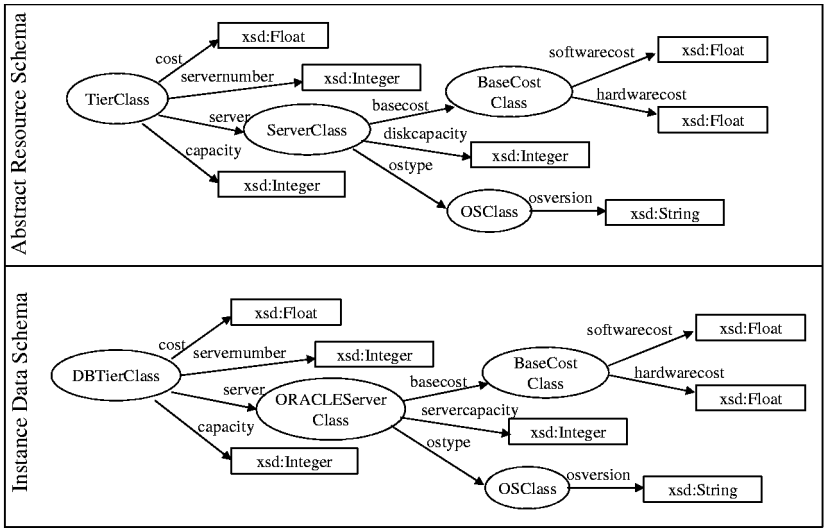


Fig. 1. A schema mapping example in a utility grid

Suppose we have the term-to-term mapping as follows: TierClass (RS) $\leftrightarrow$ DBTierClass (IS), ServerClass (RS) $\leftrightarrow$ ORACLEServerClass (IS), BaseCostClass (RS) $\leftrightarrow$ BaseCostClass (IS), OSClass (RS) $\leftrightarrow$ OSClass (IS), *diskcapacity* (RS) $\leftrightarrow$ *servercapacity* (IS). Besides, we assume that each property in RS maps to the property with the same name in IS respectively, e.g., the *cost* property in RS maps to the *cost* property in IS. In Figure 1, TierClass has properties as follows:

```
cost: Float;
servernumber: Integer;
server: ServerClass;
capacity: Integer;
```

The *cost* and *capacity* properties of TierClass can be classified as virtual properties. The value of the *cost* property is computed as $servernumber * (server.basecost.softwarecost + server.basecost.hardwarecost)$ (TierClass is omitted from the start of each parameter), where *basecost* is a property of ServerClass, $softwarecost$ and *hardwarecost* are properties of BaseCostClass. The value of the *capacity* property is computed as $servernumber * server.diskcapacity$ (TierClass is omitted from the start of each parameter), where *diskcapacity* is a property of ServerClass. We would like to capture the fact that the value of a virtual property depends on the values of the properties in its computational expression (function), e.g., the *cost* property of a TierClass resource depends upon the value of its *servernumber* property, the value S of its *server* property, the value B of the *basecost* property of S, as well as upon the values of the $softwarecost$ property and *hardwarecost* property of B.

To compute the values of virtual properties, we must acquire the values from instance data for each parameter in the function. We notice that instance data schemas may be different from abstract resource schemas, as shown in Figure 1. The functions used to compute such values may take different parameters in instance data schemas. For example, the *capacity* property of DBTierClass should be computed as $servernumber * server.servercapacity$, where *servercapacity* is a property of ORACLEServerClass instead of ServerClass. To automate the acquisition of the values for the parameters, term-to-term mappings are not enough. We facilitate the complex mappings for complete parameters in the functions of virtual properties, e.g., (TierClass.)$server$.(ServerClass.)$diskcapacity$ in RS maps to (DBTierClass.)$server$.(ORACLEServerClass.)$servercapacity$ in IS. In later sections, we describe our approach for supporting such semantics as complex mappings, algebraic relationships and dependency relationships spanning multiple data models by using semantic web technologies.

3 Representing Virtual Properties

We define a **virtual property** as a property that associates a resource with a value (a resource or a literal) that is calculated using some function, along with one or more parameter paths based upon the other properties of the source resource as well as the properties of the other resources in the paths. A **property chain** ρ is defined as a sequence of one or more properties separated by dots $p_1.p_2 \ldots p_n$, where $\forall p_k, 1 < k \leq n, p_k$ is a property of a class to which property p_{k-1}'s range is constrained. For example, *server.diskcapacity* is a property chain, where *diskcapacity* is a property of the class ServerClass, which is the range of *server*. We define a parameter path P as $C.\rho$ such that a set of values can be obtained by following chain ρ from a resource of class C. For example, given a resource R in class DBTierClass, the *capacity* property of R can be computed by invoking its function with parameter paths *R.servernumber* and *R.server.servercapacity*. Figure 2 illustrates an example parameter path $TierClass.server.basecost.softwarecost$ in RS where C is the class *TierClass* and ρ is $server.basecost.softwarecost$.

Virtual properties are powerful, especially as they span data models, and enable semantic mappings between schemas, as well as between managed and management applications in grid. However, for practical use in a real-life system, they must be managed and updated as the underlying data model evolves. Below, we propose some constructs that could be used to specify and manage the associations between virtual properties and specific functions as well as the associations between functions and parameters.

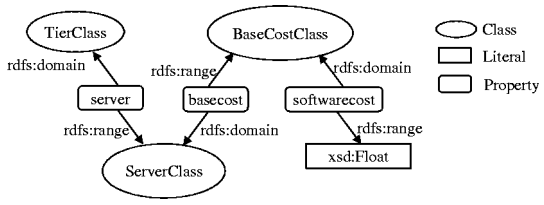


Fig. 2. Example of a Parameter Path

In order to allow multiple virtual properties to share a single function, we define a **calculated node** (Figure 3) to represent a class of resource that aggregates the parameters and other related resources associated with a functional expression. The property **hasCalculatedValue** is defined to represent the relationship between a virtual property and a calculated node. We further designate the property **hasFunction** to represent the relationship between a given calculated node and a function, and the property **hasParam** to represent the relationship between a given calculated node and an aggregation node that specifies a parameter to that node's function. The set of statements $\{(RC_u, VP_x, V_y)$, $(VP_x, hasCalculatedValue, CN_z)$, $(CN_z, hasParam, PP_v)$, $(CN_z, hasFunction, F_k)\}$ indicates that $\forall\ R_i \in RC_u$ (class), $VP_x \in VP$ (virtual properties), $CN_z \in CN$ (calculated nodes), $PP_v \in PP$ (parameters), and $F_k \in F$ (functions), resource R_i's value V_y for property VP_x can be calculated using function F_k with PP_v as one of its parameters.

Optionally, the value of a virtual property may be cached. Virtual properties may be associated with a variety of cache policies. Figure 3 predefined property "hasCachePolicy," which attaches a specific cache policy to a virtual property.

4 Managing the Evolution of Mappings

We have developed algorithms and techniques that allow us to support schema evolution operations upon classes and properties while managing the dependency relationships between resources, classes, and properties (including virtual properties). Our goal is to support both structural and semantic changes to data models while maintaining both simple (e.g., term-to-term) and complex (e.g., virtual properties) mappings between models. An example of a structural

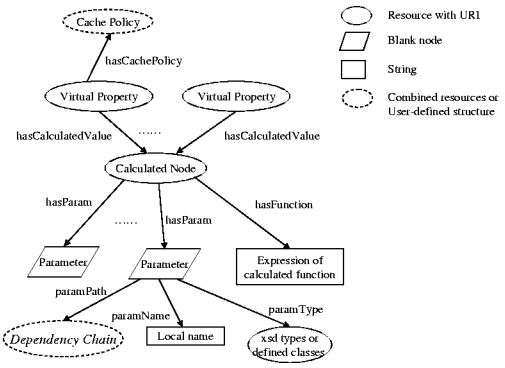


Fig. 3. Example of a Calculated Node

change would be the deletion of a property, while an example of a semantic change would be an update to a calculated node's function definition.

In this section, we define: (1) a technique for creating a special dependency chain construct to represent the dependency relationships of the parameters for each specific calculated node. (2) algorithms that use the dependency chains to create mappings for virtual properties across models, using term-to-term mappings. (3) algorithms that use the dependency chains to identify efficiently exactly the set of calculated nodes/virtual properties that are potentially affected by an update to a resource, property, or class, regardless of whether two parameters share a partial subpath.

4.1 Parameter Dependency Chains

In the context of maintaining complex mapping relationships, we capture the relationship between the function associated with a given calculated node and the resources upon which it depends (e.g., that are used in parameters to the function). Since it is not our intention to limit our methodology to a specific query or constraint language at this time, we treat the specification of the aggregation function as an opaque string.

Figure 3 sketches each parameter as composed of three parts, name, type and path, that are aggregated via a blank node — Parameter. (Normally, the type of a parameter is a data type in XML schema or a resource type defined in a domain.) By a predefined property "paramName," the local name of a parameter is attached to the corresponding Parameter node. In this way, *Calculated Nodes* cannot share parameters. We present an alternative design in Figure 4 that enables parameter reuse.

Figure 4 sketches how parameter nodes could be implemented as RDF resources instead of blank nodes. Explicit mappings between the parameter resource identifiers (URIs) and the parameter local names are provided and connected to the calculated node by a predefined property "paramMapping." For instance, in the example of Figure 6, we may map the local name "param1" to the

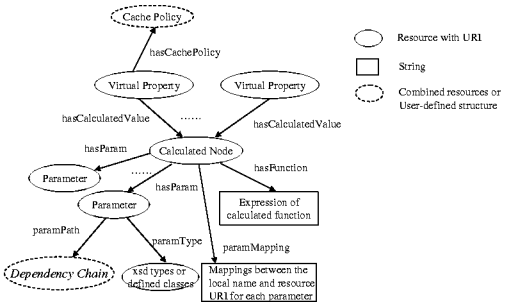


Fig. 4. Design for associating parameters with calculated nodes

parameter node of "P_servernumber" and "param2" to the node of "P_diskcapacity"[1]. This design helps calculated nodes share parameter resources.

We use the parameter paths associated with the function to construct dependency chains, which capture the dependency relationships between the calculated node and related properties in the paths. We leverage OWL's ability to specialize properties to define a special type of transitive property, **FunctionalDependency**, to model the dependency chain relationship (sketched in Figure 5). We then create a new subproperty[2] of FunctionalDependency for each dependency chain of a new calculated node.

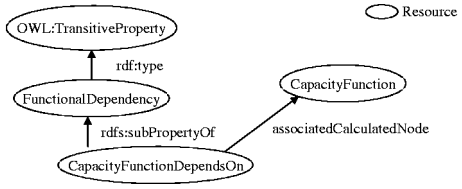


Fig. 5. CapacityFunctionDependsOn specializes the FunctionalDependency property.

The set of statements $\{(CN_x, hasParam, PP_v), (PP_v, paramPath, p_j), (p_j, p_i, p_k), (p_i, rdfs : subPropertyOf, FunctionalDependency)\}$ indicates that $\forall p_i, p_j, p_k \in P$ (properties), $PP_v \in PP$ (parameters), and $CN_x \in CN$ (calculated nodes), CN_x depends upon properties p_j and p_k, and property p_j depends upon property p_k. For example, in Figure 6, the calculated node CapacityFunction depends upon the property *server* and *diskcapacity*, and the property *server* depends upon the property *diskcapacity*. Notice that paramPath is a predefined property to connect a Parameter resource node (represented as an aggregation parameter path conceptually) to a dependency chain. Because we use OWL's subproperty

[1] The property "paramMapping" is omitted from Figure 6.

[2] We assume that we can employ some existing techniques to name a new subproperty without collisions with used names.

capabilities to perform this modelling, we can thus distinguish between different dependency chains, even when multiple *Calculated Nodes* share a given parameter path.

Example 1. Figure 6 shows the design of dependency chain for the example of $TierClass.capacity = TierClass.servernumber * TierClass.server.diskcapacity$. A calculated node "CapacityFunction" is created for the virtual property TierClass.capacity. Two parameter paths "servernumber" and "server.diskcapacity" are involved in this calculated node. For the complex parameter path, we need to express the dependency relationship between property "server" and "diskcapacity". A subproperty "CapacityFunctionDependsOn" of FunctionalDependency is created for the calculated node "CapacityFunction" to express dependency relationship.

This design is convenient to track the affected properties upon any changes in the data models. Furthermore, the subproperties are hooked back to the calculated nodes via the predefined property "associatedCalculatedNode." This makes it easy to find all of the virtual properties that would be affected by any given class or property. Notice that this design is applicable for a calculated node with multiple dependency chains, where each dependency chain has a unique subproperty of FunctionalDependency and all the subproperties are hooked back to the same calculated node.

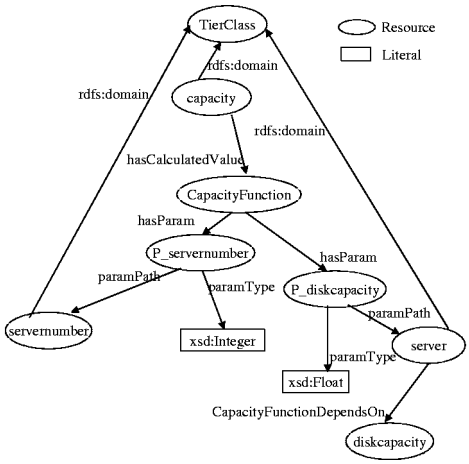


Fig. 6. Example of a dependency chain

4.2 Mapping virtual properties

Suppose that given a model of a resource type system with a virtual property and a simple term-to-term mapping from that model (e.g., RS) to a second

model (e.g., IS), we wish to calculate the value of that virtual property in terms of data belonging to the second model. For this, we need to (1) retrieve the function definition associated with the virtual property (stored in the calculated node), and (2) map the parameter paths from the first model to the second.

For simple parameters, we can directly use the term-to-term mapping to map the parameters. For complex parameters, however, we must validate each parameter path, for which we do use the dependency chain. We provide here only a general explanation of this algorithm. We first find mappings for each element of the parameter path in the term-to-term mapping. We then validate mappings for complete parameter paths to verify that we can map all parameters of the virtual property's function (see the algorithm details in appendix). If a mapping for a parameter path is not complete, we provide heuristics for discovering appropriate alternative mappings. Note that there may be multiple mappings for a parameter (simple or complex); the user selects the most appropriate one. Please refer to Section 5 for implementation details.

Validating Virtual Property Mappings Once we have built a dependency chain, we can validate whether it can be mapped to another chain. Given a dependency chain $p_1.p_2.\ldots.p_n$ in a model M, the mapped dependency chain $p'_1.p'_2.\ldots.p'_n$ in another model M' is valid if (1) given a class D_1 in the domain of p_1, one of its mapped classes, D'_1 in M', is in the domain of p'_1, the mapped property of p_1; (2) p'_2, p'_3, ..., p'_n are mapped properties of p_2, p_3, ..., p_n respectively; (3) for p'_i, where $i = 1, 2, \ldots, n-1$, its range can be matched with the domain of p'_{i+1}, i.e., a class in the range of p'_i is in the domain of p'_{i+1}.

Example 2. Figure 7 shows a complex mapping with a validated dependency chain. Suppose the dependency chain *server.basecost.softwarecost* in RS is mapped to *server.basecost.softwarecost* in IS. As shown, *server*, *basecost* and *softwarecost* in RS are mapped to *server*, *basecost* and *softwarecost* in IS respectively. DBTierClass is a class in the domain of *server* in IS that is mapped by the class TierClass which is in the domain of *server* in RS. In addition, within IS, the range of *server* is ORACLE-ServerClass, which is the domain of *basecost*, and the range of *basecost* is BaseCost-Class, which is the domain of *softwarecost*. Therefore, *server.basecost.softwarecost* in IS is a validated dependency chain of *server.basecost.softwarecost* in RS.

Heuristics for alternative mappings If there are not enough mappings between two models, we may not be able to find a validated dependency chain in a target model. The specific reasons for such failure are as follows: (1) a property has no mapped properties in the target model; (2) the domains of the first properties in two chains cannot be matched; (3) the range of a property cannot be matched with the domain of the next property in the mapped chain. In this situation, we have some heuristics to suggest new mappings between two models so that a validated dependency chain can be possibly retrieved. These heuristics can be used, for example, to suggest consistent mappings to users interactively.

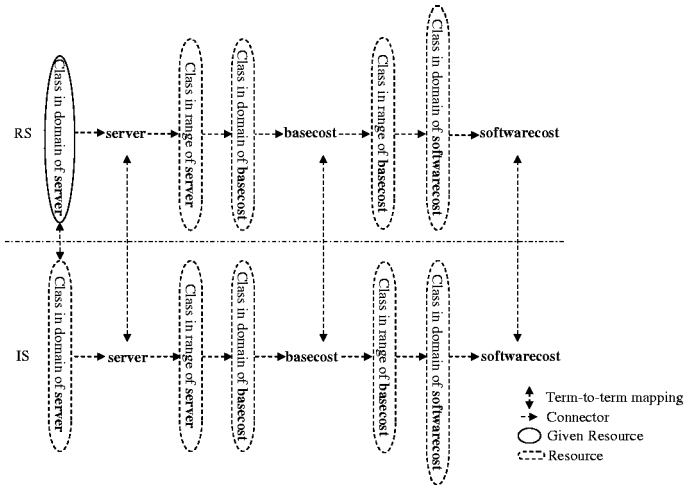


Fig. 7. Example of a validated dependency chain

Figure 8 shows three heuristics. In the second failure case where the domains of the first properties in two chains cannot be matched, e.g., as shown in the upper left of the figure, the class D_1' in the domain of p_1' (the mapped property of p_1) is not mapped by the class D_1 in the domain of p_1. However, an equivalent class C of D_1' is mapped by D_1. Therefore we may suggest a new mapping $D_1 \cong D_1'$ to the user such that the domains of p_1 and p_1' can be matched.

On the other hand, as shown in the lower left part of Figure 8, D_1 has a mapped class D_1', but D_1' is not in the domain of the property p_1', the mapped property of p_1. If D_1' has a property p_{11}', whose range can be matched with the domain of p_2', the next property in the dependency chain, then we may suggest a new mapping $p_1 \cong p_{11}'$.

In the third failure case where the range of a property cannot be matched with the domain of the next property in the target chain, for example, as shown in the right part of the figure, p_1' and p_2' are two neighbor properties in the dependency chain, while the range of p_1' cannot be matched with the domain of p_2'. If a class R_1' in the range of p_1' has a property p_{21}', whose range can be matched with the domain of p_3', the next property of p_2' in the chain, we may suggest a new mapping $p_2 \cong p_{21}'$ such that the mapped chain becomes $p_1'.p_{21}'.p_3'$. In the first failure case where a property has no mapped properties in the target model, the problem may be solved by the new suggested mappings of the above heuristics.

Methodology for Maintaining Mappings We propose a methodology for maintaining both simple and complex mappings, especially the dependency relationships. This methodology covers several aspects listed as follows. First, when a calculated node is created, the path expression of each parameter is parsed

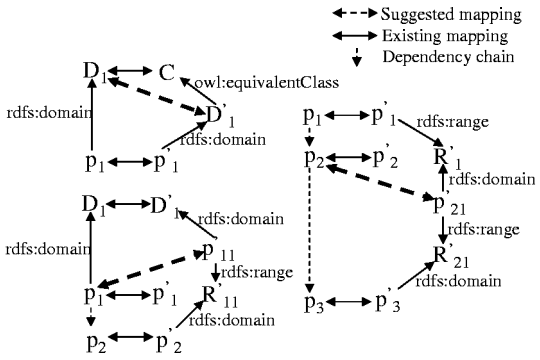


Fig. 8. Heuristics for suggesting mappings

and built as a dependency chain connected with a specific sub-property of FunctionalDependency for that chain with that calculated node. Second, if a path expression is redefined, the corresponding calculated node is going to be rebuilt. Then the virtual properties sharing that calculated node are to be reported to user. It is up to the user for checking whether a virtual property can be associated with an updated calculated node. Third, if a property in a class is deleted or redefined such that a dependency chain including that property is not valid according to the semantic constraint, all the calculated nodes having the invalid dependency chains are to be retrieved and based on this information, all the virtual properties related to the nodes are to be reported to the user. It is still up to the user deciding how to adapt those virtual properties.

For instance, in the example of TierClass.capacity (which is computed using the function $TierClass.servernumber * TierClass.server.diskcapacity$), if the property *diskcapatiy* is deleted or redefined, the dependency chain *server.diskcapacity* may not be valid any more. The virtual property TierClass.capacity is affected by these changes. Once notified, the user can quickly identify the affected elements in the system and decide how to adapt.

Basic Operations for Maintaining Mappings Given a change, three classes of affected objects are to be identified: dependency chains, calculated nodes (that depend upon the affected path expressions or else directly upon the affected property) and virtual properties (that use the affected calculated nodes). Generally, four basic structural change operations are to be considered: (1) a property has a new domain or range class added; (2) a property has a domain or range class deleted; (3) a property is deleted; (4) a new property is added. All other changes (class gains property, property changes domain, etc.) can be expressed using these operations. If a new property is added, then existing path expressions are not affected (because RDF does not allow properties to be overridden). If a property has gained a new domain or range class, C, or loses its domain or range relationship with C, we need to check whether some existing path ex-

pressions (property chains) become invalid due to this change. If a property is deleted, then the related path expressions are definitely to be affected and we must identify the affected virtual properties.

For the case (3) where a property is deleted, we can identify affected virtual properties in two ways. If the updated property p happens to be the first property in a dependency chain, we can retrieve the affected virtual properties with a query (written in RDQL [21]) as follows:

```
SELECT ?vp
WHERE (?vp, <ns1:hasCalculatedValue>, ?cn),
      (?cn, <ns1:hasParam>, ?pm),
      (?pm, <ns1:paramPath>, <ns1:p>)
USING ns1 FOR <http://www.hpl.hp.com/Demo#>
```

This query is from our prototype implementation (See Section 5). Notice that the properties $hasCalculatedValue$, $hasParam$ and $paramPath$ are predefined constructs introduced in Section 3 and Section 4.1. If p is not the first property, we use dependency chains. The first step is to find all the properties $?x$ that depend on p. Since properties in dependency chains are connected with sub-properties of $FunctionalDependency$, we can then find those sub-properties $?subProp$ which connect $?x$ to p. These two steps can be done with a query as follows:

```
SELECT ?subProp
WHERE (?x, ?subProp, <ns1:p>),
      (?subProp, <rdfs:subPropertyOf>, <ns1:FunctionalDependency>)
USING ns1 FOR <http://www.hpl.hp.com/Demo#>,
      rdfs FOR <http://www.w3.org/2000/01/rdf-schema#>
```

If such $?subProp$ exists, we can find the unique calculated node associated with $?subProp$ via the predefined property $associatedCalculatedNode$. Finally, the related virtual properties can be retrieved.

For the case (1) where a property p has a new domain or range class added, the domain or range of p may become empty due to the conjunctive semantics for domains or ranges in RDF[16]. If that happens to p, all the paths related to p will be invalid and we must identify them.

In case (2), if a property p has a range class C deleted, we need to check, for a specific property chain ρ, whether any other class C' in the range of p is in the domain of the next property in ρ. If that happens, ρ is still valid, otherwise we should report ρ as an affected path. The check can be done in two steps. First, we need to find all the property chains in which p depends on some other property as follows:

```
SELECT ?subProp
WHERE (<ns1:p>, ?subProp, ?x),
      (?subProp, <rdfs:subPropertyOf>, <ns1:FunctionalDependency>)
USING ns1 FOR <http://www.hpl.hp.com/Demo#>,
      rdfs FOR <http://www.w3.org/2000/01/rdf-schema#>
```

If the query result is not empty, for each sub-property $subP$ of $FunctionalDependency$, we check whether the chain based on $subP$ is valid as follows:

```
SELECT ?c
WHERE (<ns1:p>, <rdfs:range>, ?c),
      (<ns1:p>, <ns1:subP>, ?nextP),
      (?nextP, <rdfs:domain>, ?c)
USING ns1 FOR <http://www.hpl.hp.com/Demo#>,
      rdfs FOR <http://www.w3.org/2000/01/rdf-schema#>
```

If such $?c$ exists, the chain based on that specific $subP$ is still valid. The same idea is applicable to the situation when p has a domain class deleted. Differently, we need to check the domain of p with the range of the previous property instead of the domain of the next property in a given chain ρ.

Example 3. Figure 9 shows a new instance data schema to be added into the utility grid in Figure 1 to replace the old instance schema. The class ORACLEServerClass is replaced by SQLServerClass, which has the property *servercost* instead of *basecost* and the property *dbcapacity* instead of *servercapacity*. These changes will affect the virtual properties *cost* and *capacity*. In addition, the mappings between RS and IS are also to be affected. One of the dependency chains in RS, $server.basecost.softwarecost$, is not mapped to $server.basecost.softwarecost$ in IS any more since the property *basecost* in IS is deleted. Suppose we know that SQLServerClass in new IS is mapped by ServerClass in RS, but we have no idea about the relationship between *servercost* in IS and *basecost* in RS. According to the third heuristics in Section 4.2, we may suggest a new mapping $basecost \cong servercost$ such that the chain $server.basecost.softwarecost$ in RS can be mapped to $server.servercost.softwarecost$ in new IS.

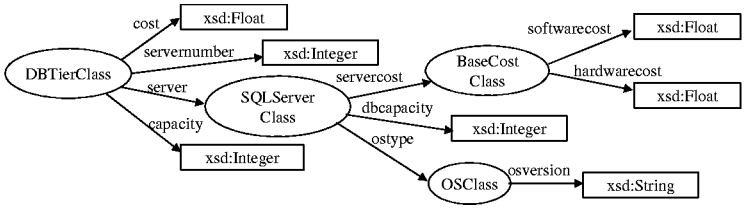


Fig. 9. A new instance data schema in the utility grid

5 Reference Implementation

Increasingly there is a need to provide resources on demand. HP's Utility Data Center, IBM's on-demand initiative, much of the work that is being done in the Grid community are utility computing initiatives. Our initial application scenario was to provide data integration capabilities for a resource provisioning system in the context of a utility data center. The resource provisioning system is responsible for the provisioning, allocation, and assignment of resources in the GRID [22]. The resource provisioning system accepts GRID requests for

resources, which represent resource types, composition specifications, policies, and time constraints. The reservation system validates each request in terms of its type and instance inventory, verifying that the request is for a valid system and that resources are available. It then reserves the appropriate resources for the specified time. Later, when the reservation matures, the assignment system re-validates that the appropriate resource instances are available, translates the request into concrete system descriptions, and passes these to the operational system for deployment. The operational system may also notify the provisioning system when the assigned resources need adjusting.

Figure 1 shows how the resource provisioning system may translate client requests for resources into specific configuration of grounded resources, adding new types of resources and evolving existing resource types. In the context of this application scenario, we performed four steps:

1. Created an OWL ontology to represent each model to be integrated.
2. Created initial mappings between the models (expressed using OWL).
3. Modelled the composite dependencies between properties and integrate them into the mappings.
4. Provided a reference implementation of a set of operations for evolving the ontologies while maintaining the mappings, using the techniques described in this paper.

Our prototype provides a tool that facilitates the management of simple and complex mappings that may span ontologies. This reference implementation was built using Jena, a Java-based open source semantic web toolkit from HP [12]. The latest version of Jena (Jena2) includes a reasoner subsystem that implements a generic rule based inference engine, as well as configured rule sets for RDFS and OWL-Lite. Our implementation extends the Jena2 Java classes that support ontology-aware models with methods that facilitate the creation of mappings and that enable the evolution of classes, models, and properties.

Figure 10 shows the architecture of our prototype, which contains two tools, an Ontology Evolution Manager and a Mapping Manager, which were implemented using three modules–an Impact Computation Engine, a Virtual Property Handler, and a Mapping Heuristics Engine. The RDF/OWL Interpreter layer provides fundamental functionalities for processing ontologies.

Both the Ontology Evolution Manager and the Mapping Manager take as input a source ontology, a target ontology, and a mapping between them. The Ontology Evolution Manager takes as additional input a specification of a proposed change to the source ontology, and returns as output a set of elements that are potentially impacted by the proposed change, a set of new dependency chains (based on the proposed change), and a set of suggested changes to the mapping. The Mapping Manager takes as additional input an identifier for a virtual property in the source ontology, and returns a mapping to the target ontology for the parameter paths of the virtual property.

The Impact Computation Engine parses change specification and identifies the impacted elements from the source ontology. It interacts with the Virtual

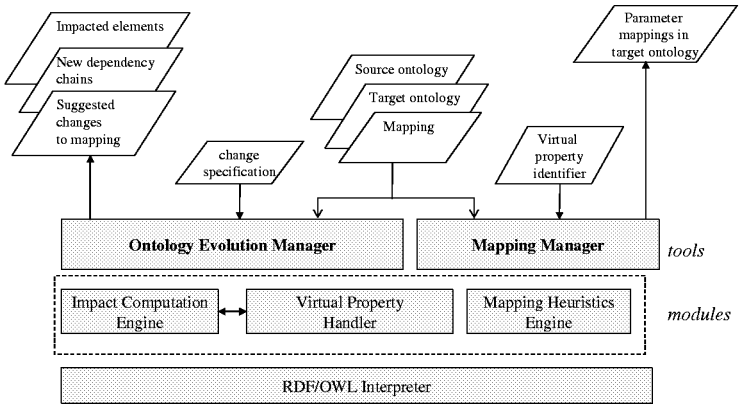


Fig. 10. Prototype Architecture

Property Handler, which identifies the impacted virtual properties (virtual properties whose parameters involve impacted elements). In addition, the Virtual Property Handler helps to construct the mappings to the target ontology for parameter paths. The third module, Mapping Heuristics Engine, is responsible for applying heuristics to suggest changes to the mappings (simple or complex).

Figures 11 and 12 are screen shots from our prototype upon the scenario in Figure 1. Figure 11 illustrates RS (on the left), IS (on the right) and the term-to-term mappings between them (in the middle). Figure 12(a) illustrates two mapped dependency chains (parameter paths), where the left one is from RS and the right one is from IS. Figure 12(b) shows the impacted virtual properties (in bold face) upon the deletion of the properties *basecost* and *servercapacity* when the old IS in Figure 1 is replaced by the new IS in Figure 9.

6 Related Work

Automatic mapping discovery. Doan et al [8] build a generic ontology matching solution using machine learning techniques as well as commonsense knowledge and domain constraints. They focus on improving matching accuracy between ontologies instead of maintaining mappings upon the changes of ontologies. Their work has not involved attributes or properties. Lopez et al [13] provide a tool that partially automates the creation of a mapping between two ontologies expressed in OWL, using string matching of labels and subclass relationships. They do not leverage more sophisticated OWL constructs, such as sub-property relationships, they do not support complex mappings such as the virtual properties we describe here, nor do they specifically address the incremental maintenance of a mapping over the evolution of the underlying ontologies. Readers who are interested in automated matching are also referred to the survey of approaches to automatic schema mapping by Rahm and Bernstein [20].

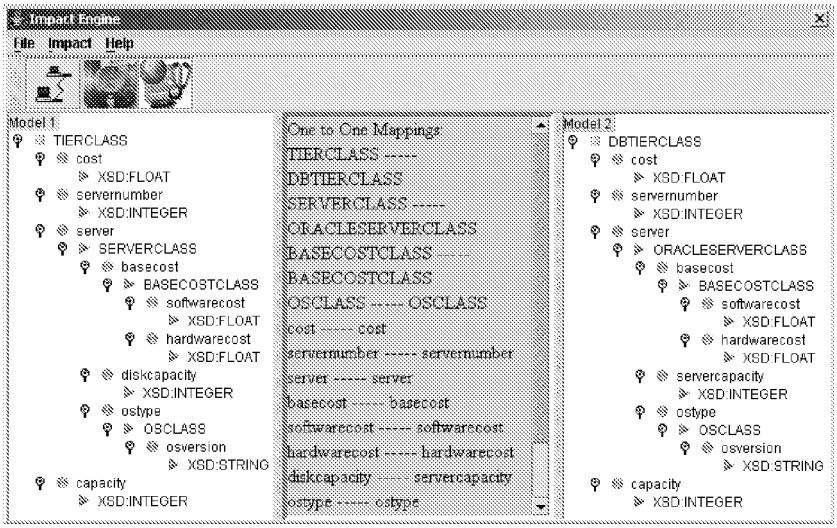
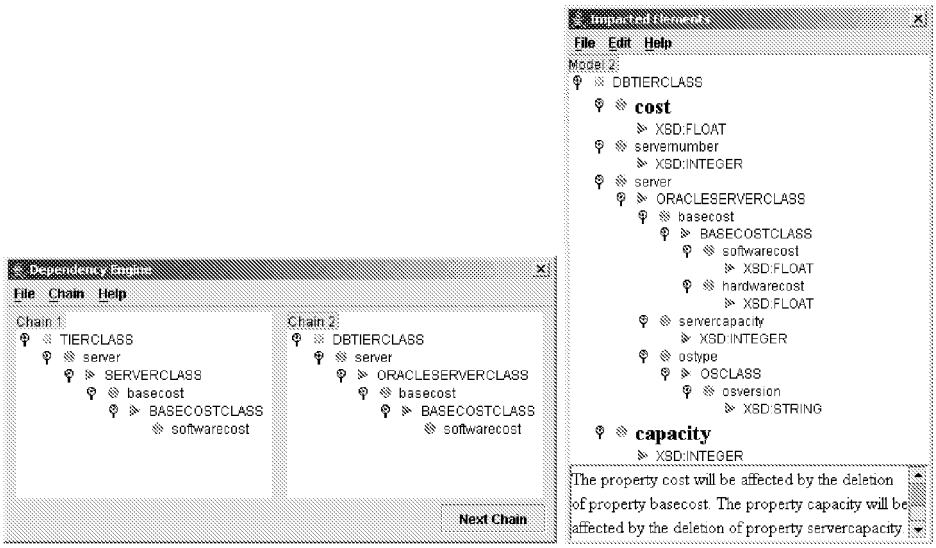


Fig. 11. Screen shot of Impact Engine



(a) (b)

Fig. 12. (a) Screen shot of mapped dependency chains, (b) Screen shot of impacted elements

Noy et al [19] propose an algorithm Anchor-PROMPT to use a set of heuristics to analyze non-local context and determine additional possible points of similarity between ontologies. These points of similarity can be further used to

suggest missing mappings between given ontologies. However, their work still concentrates on mappings between classes, not properties.

Ontology-based data integration. Wiederhold et al [28, 18] at Stanford University have proposed an ontology algebra for manipulating ontologies from different domains. They represent ontologies with a graph-based model and provide a set of operators to articulate multiple ontologies into a unified ontology using both logical rules and funtional rules. However, their goal is to answer user queries reliably and their work does not involve maintaining mappings between ontologies upon any changes. Bonatti et al [2] propose the notion of an ontology extended relation and extend relational algebra to query ontology associated relations. They provide formal definition and methodology for merging ontologies under interoperation constraints. However, this work also does not involve mapping maintenance and does not recombine property values functionally.

Model management. Melnick et al [17] identify a generic platform for model management and define an algebra to manipulate models and mappings. They address the problems about matching models, merging models, selecting and extracting a subset of a model. They also propose generic mechanisms to maintain the mappings between models. But their work does not involve properties and composite relationships among properties as described here.

Stojanovic and Motik [24] discuss how ontology editors can support consistent ontology evolution. They identify a set of requirements for ontology editors to support the alteration of an ontology while maintaining its consistency, but do not address the problem of complex mappings. Maedche et al [14] present an infrastructure to support the discovery, re-use, and evolution of distributed ontologies. Their infrastructure consists of an ontology registry, mechanisms enabling the re-use of distributed ontologies (including a mechanism for including distributed ontologies by reference into ontology languages and tools), and methods supporting the evolution of distributed (replicated) ontologies. They require the re-used ontology to be replicated in its entirety, and thus do not focus on the direct or complex mappings that we address.

Service dependency and impact management. Ensel et al [9] use RDF to create dependency graphs for tracking application service dependencies. However, they do not consider maintenance of the graph over time. Because they do not use the sub-property relationship, they cannot distinguish between separate paths through the graph.

Stuckenschmidt et al [25] address the problem of guaranteeing the integrity of a modular ontology in the presence of local changes. They present their approach to ontology modularization and propose a mechanism to detect changes and analyze their impact of the concept hierarchy. However, their work focuses on classes instead of complex mappings related to properties.

Virtual properties. Christophe Blanchi's work with DOI allows code blocks to be associated with properties, but the output of the code blocks are not integrated into the metadata. As such, the calculated data is "second-class." MIT's Haystack project has an inference engine (Adenine), which includes the ability

to set metadata using a custom language. However, this system is restricted to off-line use, which means that data may be in an inconsistent state.

A number of researchers have investigated the topic of views for the Semantic Web [27, 15]. Because Semantic Web technologies are still emerging, existing work on RDF/S views focuses on the view definition language. [27] extend RQL [3] with view primitives, and implement a view mechanism that supports view definitions that span ontologies. [15] have developed their own view definition lanaguage for RDF/S, that supports both virtual resource descriptions and virtual RDF/S schemas. Similarly, some work has addressed view creation over semi-structured data. For example, [4] supports the definition of views over XML documents using XQL queries and [29] considers the maintenance problem of graph structured views.

Our simple mappings could be considered a kind of RDF/OWL view, and our work on mapping evolution could be applied to the Semantic View systems proposed by the other researchers. However, the complex mappings that we address are fundamentally different from views, in that they are not specified using a view definition language. The disadvantage of this is that our mappings thus do not qualify as views in the traditional sense. The advantage is that separating between the parameters and the function definitions allows us to support mapping between different ontologies in an extremely flexible manner. For example, multiple virtual properties can be associated with a common calculated node, and multiple calculated nodes can share parameter path definitions. This minimizes the mappings that must be made when integrating models; instead of having to recompute the mapping for each virtual property, we can reuse existing mappings. Similarly, should a calculated node's function be updated to take different parameters due to evolution or integration, we can update just the affected parameter specifications. In addition, our work is unique for our focus on evolution, which to the best of our knowledge nobody else considers in the context of RDF views.

7 Conclusions / Future Work

Web services enable today's enterprises to combine a plethora of loosely-coupled distributed systems. One of the fundamental hard problems raised by this scenario is how to enable semantic interoperability between related components-especially when the relationships span layers of the enterprise. Our vision is to leverage Semantic Web technologies to provide an extremely lightweight semantic modeling layer that enhances existing methods for integrating data and services. This additional semantic layer allows us to incorporate existing semantic information into the integration process.

Because enterprises need to accommodate continually evolving infrastructures, technologies, and available services, we believe that support for the efficient evolution of mappings is just as important as the initial creation of mappings. We identify two kinds of mappings–simple term-to-term mappings and complex, functionally-derived, mappings. We believe that Semantic Web technologies such

as RDF/OWL are well-suited for maintaining such mappings because of the Semantic Web's emphasis on flexibility and extensibility, and because of its natural support for inference.

We have presented here techniques and algorithms for representing and evolving simple and complex mappings between ontologies. We have introduced virtual properties (properties whose values are functionally derived from the values of other properties), and described how these can be maintained as the underlying data models evolve. We can support both structural and semantic changes to data models while maintaining both simple (e.g., term-to-term) and complex (e.g., virtual properties) mappings between models. We have implemented a prototype demonstrating our methodology in an application scenario from the management domain.

We believe that virtual properties represent a promising technology for data integration. We thus hope to extend our work in the application scenario by integrating with additional management systems, as well as by performing studies to evaluate the effectiveness of our methods.

We also plan to focus next on the specification languages used to define the function associated with a given virtual property, and to apply the principals of this work to the integration of constraints and policies. Furthermore, we may consider model validation for new composed resource types, which is to validate newly specified resource types in terms of pre-existing constraints and policies. For example, we could apply our techniques to the problems of root cause analysis or business impact analysis.

Acknowledgements. We thank Bernard Burg, Patrick Goldsack, Vijay Machiraju, Mike Short, Sharad Singhal, V. S. Subrahmanian, and Aad Van Moorsel for their support and guidance. We are grateful to the Jena team for the Jena Semantic Web Framework for Java. We thank our reviewers at SemPGRID04 for their extremely valuable feedback. We especially thank Dave Reynolds for his thoughtful comments and suggestions, and Akhil Sahai for many wonderful discussions and for his help and guidance regarding the Grid resource model.

References

1. F. Baader, I. Horrocks, and U. Sattler. Description logics for the semantic web. *KI – Künstliche Intelligenz*, 16(4):57–59, 2002.
2. P. A. Bonatti, Y. Deng, and V. S. Subrahmanian. An ontology-extended relational algebra. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IEEE IRI)*, 2003.
3. J. Broekstra. Sesame RQL: a Tutorial. http://www.openrdf.org/publications/rql-tutorial.html.
4. L. Chen and E. A. Rundensteiner. Aggregate path index for incremental web view maintenance. In *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems*, pages 231–238, 2000.
5. David De Roure, N. Jennings, and N. Shadbolt. The Semantic Grid: A Future e-Science Infrastructure. In F. Berman, Anthony J.G. Hey, and G. Fox, editors,

Grid Computing: Making the Global Infrastructure a Reality, pages 437–470. John Wiley and Sons, 2003.
6. M. Dean and G. Schreiber. OWL Web Ontology Language Reference. W3C Recommendation, February 2004. http://www.w3.org/TR/owl-ref/.
7. Y. Deng, H. Kuno, and K. Smathers. Managing the evolution of simple and complex mappings between loosely-coupled systems. In *The Second Workshop on Semantics in Peer-to-Peer and Grid Computing at the Thirteenth International World Wide Web Conference*, May 2004.
8. A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Halevy. Learning to match ontologies on the Semantic Web. *VLDB Journal, Special Issue on the Semantic Web*, 2003.
9. C. Ensel and A. Keller. An approach for managing service dependencies with xml and the resource description framework. *Journal of Network and Systems Management, Special Issue on Selected Papers of IM 2001*, 10(2):27–34, June 2002.
10. Global Grid Forum. Open Grid Services Infrastructure (OGSI), 2003.
11. C. Goble and David De Roure. The grid: An application of the semantic web. *ACM SIGMOD Record: Special section on semantic web and data management*, 31(4), December 2002.
12. Jena 2 - A Semantic Web Framework, 2003. http://www.hpl.hp.com/semweb/jena2.htm.
13. J. B. Jorge E. Lopez de Vergara, Victor A. Villagra. An ontology-based method to merge and map management information models. In *HP Openview University Association Tenth Plenary Workshop*, Geneva, Switzerland, July 2003.
14. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. In *Proceedings of the twelfth international conference on World Wide Web*, pages 439–448, 2003.
15. A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the semantic web through rvl lenses. In *Proceedings of the Second International Semantic Web Conference*, October 2003.
16. F. Manola and E. Miller. RDF primer, W3C recommendation. http://www.w3.org/TR/rdf-primer/, February 2004.
17. S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *ACM SIGMOD 2003*, San Diego, CA, June 2003. citeseer.nj.nec.com/melnik03rondo.html.
18. P. Mitra, G. Wiederhold, and M. Kersten. Graph-oriented model for articulation of ontology interdependencies. In *Proceedings of the 7th International Conference on Extending Data-base Technology*. Springer-Verlag, 2000.
19. N. F. Noy and M. A. Musen. Anchor-prompt: Using non-local context for semantic matching. In *Workshop on Ontologies and Information Sharing at the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 2001.
20. E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10:334–350, November 2001.
21. RDQL - RDF Data Query Language. http://www.hpl.hp.com/semweb/rdql.htm.
22. A. Sahai, S. Singhal, R. Joshi, and V. Machiraju. Automated policy-based resource construction in utility computing environments. Technical Report HPL-2003-176, Hewlett-Packard Laboratories, August 2003.
23. M. Short. Are you ready for enterprise systems that fix themselves? *DevX.com*, November 2002. http://www.devx.com/enterprise/Article/9891.
24. L. Stojanovic and B. Motik. Ontology evolution within ontology editors. In *Proceedings of the OntoWeb-SIG3 Workshop at the 13th International Conference*

on Knowledge Engineering and Knowledge Management (EKAW) , pages 53–62, September 2002.
25. H. Stuckenschmidt and M. Klein. Integrity and change in modular ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'03*, 2003.
26. F. van Harmelen. OWL: an ontology language for the Semantic Web. In *SIKS / IKAT Symposium: Brain, Language, and Artificial Intelligence*, Universiteit Maastricht, June 2003. http://www.cs.unimaas.nl/SIKSsymp/harmelen.pdf.
27. R. Volz, D. Oberle, and R. Studer. Implementing views for light-weight web ontologies. In *Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS'03), July 16 - 18, 2003, Hong Kong, SAR*, pages 160–170. IEEE Computer Society, 2003.
28. G. Wiederhold. An algebra for ontology composition. In *Proceedings of 1994 Monterey Workshop on Formal Methods*, pages 56–61, September 1994.
29. Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *IEEE Int. Conf. on Data Engineering*, pages 116–125, 1998.

Appendix A: Algorithms w.r.t. dependency chains

Figure 13 presents the CMA algorithm (short for **Create Complex Mapping Algorithm**) to create mappings for virtual properties across schemas.

algorithm CMA(S, T, M, P, C)
/* Input: source schema S, target schema T, mapping ontology M, dependency chain P, class C */
/* Output: a set V of validated dependency chains in T for P */
1) $D \leftarrow$ the mapped classes for C in T via M
2) $X_{curr} \leftarrow \emptyset$, $L \leftarrow \emptyset$, $l \leftarrow \emptyset$
3) $p_{curr} \leftarrow$ the first property of P
4) $X \leftarrow$ the mapped properties for p_{curr} in T via M
5) for each property $p \in X$
6) for each class $d \in D$
7) if (d is a domain class of p) then $l \leftarrow l \cup (d, p)$
8) if ($l == \emptyset$) then return *null*
9) $L \leftarrow L$ appended by l, $l \leftarrow \emptyset$
10) $b \leftarrow$ TRUE
11) while (P has new properties)
12) $p_{curr} \leftarrow$ the next property of P
13) $X_{curr} \leftarrow$ the mapped properties for p_{curr} in T via M
14) for each property $p \in X$
15) $D \leftarrow$ the range classes of p in T
16) for each property $p' \in X_{curr}$
17) for each class $d \in D$
18) if (d is a domain class of p') then $l \leftarrow l \cup (p, p')$
19) if ($l == \emptyset$) then
20) $b \leftarrow$ FALSE
21) break
22) $L \leftarrow L$ appended by l, $l \leftarrow \emptyset$, $X \leftarrow X_{curr}$
23) if ($b ==$ FALSE) then return null
24) $V \leftarrow$ ConstructChain(L)
25) return V

Fig. 13. Create Complex Mapping Algorithm **CMA**

In line 24, ConstructChain(L) is a subroutine to connect pairs from the layers in L such that each chain contains n pairs, where each pair (x_{i_1}, x_{i_2}) is from each layer l_i respectively, i=1,2,...,n. In addition, for two neighbor pair (x_{i-1_1}, x_{i-1_2}) and (x_{i_1}, x_{i_2}) in a chain, $x_{i-1_2} = x_{i_1}$, where $1 < i \leq n$. Notice that in each chain, the first element x_{1_1} of the fist pair is a class (a domain class of x_{1_2}), while other elements are properties.

Figure 14 presents the IPA algorithm (short for **Identify Impacted Properties Algorithm**) to identify the set of virtual properties that are potentially

affected by an update to a property p with a specific domain class C in a schema S.

algorithm IPA(S, p, C)
/* Input: schema S, property p, class C */
/* Output: a set V of virtual properties to be affected */
1) $V \leftarrow \emptyset$, $F \leftarrow \emptyset$
2) $N \leftarrow$ the calculated nodes which have a parameter path where p is the first property
3) for each node $n \in N$
4) $V \leftarrow V \cup$ the virtual properties connected to n
5) $X \leftarrow$ the properties that depend on p
6) for each property $x \in X$
7) if (C is a range class of x) then
8) $F \leftarrow F \cup$ the properties that connect x to p
9) for each property $f \in F$
10) $n \leftarrow$ the calculated node associated with f
11) $V \leftarrow V \cup$ the virtual properties connected to n
12) return V

Fig. 14. Identify Impacted Properties Algorithm **IPA**

Appendix B: Some comments comparing RDF with the Object-Oriented paradigm

The RDFS core properties related to type relationships are:

- rdfs:subClassOf - A transitive subset/superset relationship indicating c lass membership.
- rdfs:subPropertyOf - A transitive subset/superset relationship indicating property specialization.
- rdf:type - Indicates membership of a class.

For example, if an RDF data store contains statements:

1. parentOf rdfs:subPropertyOf ancestorOf
2. PersonA parentOf PersonB

then an RDF/OWL/RDF Schema system would support the inference that: "PersonA ancestorOf PersonB" and thus a query selecting resources in a ancestorOf relationship with PersonA would include PersonB in its results.

Unlike the object-oriented model, where class membership dictates whether objects can gain and or lose properties, in the property-centric RDF paradigm resources gain and lose class memberships based on their properties. An object-oriented system would associate properties with classes; instances would then acquire properties by acquiring the type of the class. Similarly, the rdf:type property can be used to give a resource class membership. RDFS, however, is property-centric, and the classes to which a resource belongs also depend on the properties for which the resource has values (via statements). RDFS defines constraints that specify the domain and range properties with which resources can be associated. The basic RDFS constraint properties are:

- rdfs:range - A range class for a property type. That is, resources assigned as values for the property have the type of the given range class.
- rdfs:domain - A domain class for a property type. That is, resources that have values for the property have the type of the given domain class.

If a constraint specifies that a class is in the domain of a given property, any resource that is associated with that property is a member of the specified class. Similarly, if a constraint specifies that a class is in the range of a given property, then any resource that is assigned as a value for that property is a member of the specified class. In addition, a new resource can obtain membership in a given class via the rdf:type property even if it lacks any other property assignments. That is to say, there are three ways that a resource can be associated with a given class: implicitly, via a domain or a range constraint, or explicitly via a rdf:type assignment. For example, Figure 15 shows some statements about the types and constraints of classes ClassA, ClassB, and ClassC. ClassC is designated a subclass of ClassB. PropA has ClassA in its domain, PropC has ClassC in its domain, and PropA has ClassB in its range.

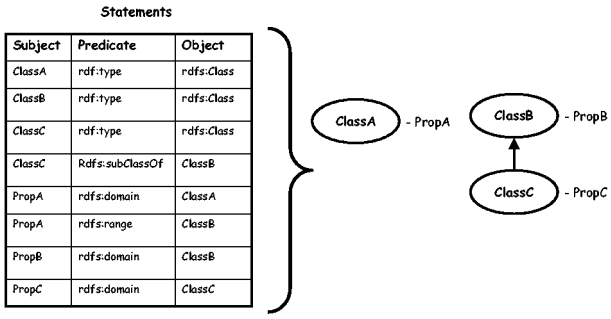


Fig. 15. RDFS extends RDF with a type hierarchy.

If we now extend the statements in Figure 15 with statements in Figure 16, we see that Resource1 is the subject of a statement with PropA as a property, and is thus implicitly a member of ClassA (because ClassA is in the domain of PropA); Resource2 is the object of a statement with PropA as a property, and is thus implicitly a member of ClassB (because ClassB is in the range of PropA); and Resource3 is the subject of a statement assigning it the type of ClassC and is thus explicitly a member of ClassC and implicitly a member of ClassB (because ClassC is a subclass of ClassB).

Statements

Subject	Predicate	Object
Resource1	PropA	Resource2
Resource1	PropB	valueY
Resource2	ProbC	valueZ
Resource1	PropC	valueZ
Resource3	rdf:type	ClassC

Fig. 16. Associating resources with types in RDFS.

Fundamental differences between RDFS and an object-oriented model include:

- Because properties are unique objects, RDFS does not support the overriding of properties.
- RDFS supports extremely dynamic reclassification in which resources gain and drop membership in classes implicitly by changing their properties.

- RDFS does not require resources to have values defined for any particular properties, regardless of their class membership. All a resource need in order to belong to a given class is to be associated with one property that lists the class or one of its subclasses in its domain constraint.
- The RDFS type hierarchy does not require a single point of inheritance. E.g., two classes that have the same property may not have inherited it from a common source.
- Because the rdfs:subPropertyOf property is independent from the rdfs:subClassOf property, an RDFS type hierarchy can include a property that is a subProperty of another property without requiring that that the subProperty be associated with a class that is a subclass of a class associated with the superProperty.
- RDFS does not require properties to have either domain or range constraints. RDFS thus allows resources to have values for properties not associated with any class.