A Component-Based Framework For Ontology Evolution

Michel Klein
Vrije University Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam, The Netherlands

Natalya F. Noy
Stanford Medical Informatics,
Stanford University,
Stanford, CA 94305

Abstract

Support for ontology evolution becomes extremely important in distributed development and use of ontologies. Information about change can be represented in many different ways. We describe these different representations and propose a framework that integrates them. We show how different representations in the framework are related by describing some techniques and heuristics that supplement information in one representation with information from other representations. We present an ontology of change operations, which is the kernel of our framework.

1 Support for Ontology Evolution

Ontologies are increasing in popularity, and researchers and developers use them in more and more application areas. Ontologies are used as shared vocabularies, to improve information retrieval, or to help data integration. Neither the ontology development itself nor its product—the ontology—is a single-person enterprise. Large standardized ontologies are often developed by several researchers in parallel (e.g. SUO[1] [9]); a number of ontologies grow in the context of peer-to-peer applications (e.g. Edutella [5]); other ontologies are constructed dynamically [2]. Successful applications of ontologies in such uncontrolled, de-centralized and distributed environments require substantial support for change management in ontologies and ontology evolution [7].

Given an ontology O and its two versions, V_{old} and V_{new}, a complete support for change management in an ontology environment includes support for the following tasks.[2]

Data Transformation: When an ontology version V_{old} is changed to V_{new}, data described by V_{old} might need to translated to bring it in line with V_{new}. For example, if we merge two concepts A and B from V_{old} into C in V_{new}, we must combine instances of A and B as well.

Data Access: Even if data is not being transformed, if there exists data conforming to V_{old}, we often want to access this data and interpret it correctly via V_{new}. That is, we should be able to retrieve all data that was accessible via queries in terms of V_{old} with queries in terms of V_{new}. Furthermore, instances of concepts in V_{old} should be instances of equivalent concepts in V_{new}. This task is a very common one in the context of the Semantic Web, where ontologies describe pieces of data on the web.

Ontology Update: When we adapt a remote ontology to specific local needs, and the remote ontology changes, we must propagate the changes in the remote ontology to the adapted local ontology [8].

Consistent Reasoning: Ontologies, being formal descriptions, are often used as logical theories. When ontology changes occur, we must analyze the changes to determine whether specific axioms that were valid in V_{old} are still valid in V_{new}. For example, it might be useful to know that a change does not affect the subsumption relationship between two concepts: if $A \sqsubseteq B$ is valid in V_{old} it is also valid in V_{new}. While a change in the logical theory will always affects reasoning *in general*, answers to specific queries may remain unchanged.

Verification and Approval: Sometimes developers need to verify and approve ontology changes. This situation often happens when several people are developing a centralized ontology, or when developers want to apply changes selectively. There must be a user interface that simplifies such verification and allows developers to accept or reject specific changes, enabling execution of some changes and rolling back of others.

This list of tasks is not exhaustive. The tools that exist today support these tasks in isolation. For example, the KAON framework [10] supports *evolution strategies*, allowing developers to specify strategies for updating data when changes in an ontology occur. The SHOE versioning system specifies which versions of the ontology the current version is *backward compatible* with [3]. Many ontology-editing environments (e.g., Protégé [1]) provide logs of changes between versions. While these tools support some of the ontology-evolution tasks, there is no interaction or sharing of information among the tools. However, many of these tasks require the same elements in the representation of change. Imple-

[1] `http://suo.ieee.org/`

[2] Note that V_{new} is not necessarily a unique replacement for V_{old}. There might be several new versions based on the old version, and all of them could exist in parallel. The labels are just used to refer to two versions of an ontology where V_{new} has evolved from V_{old}.

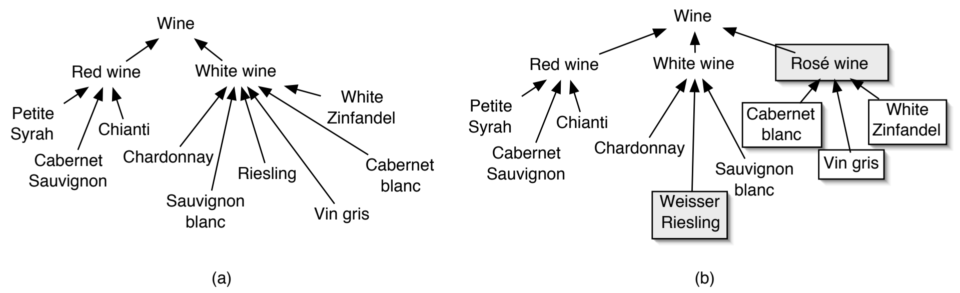


Figure 1: *Two versions of a wine ontology (a and b).*

mentation of support for one task can, and should, use the change information acquired for another, rather than try to determine it from scratch. Having a general framework for ontology evolution that allows tools supporting different evolution tasks to share change information and leverage change information obtained by other tools, will make ontology evolution much more efficient.

In this paper, we propose a framework that integrates several sources of information about ontology change. We explain how these different change representations are related to one another and show how we can supplement information in one representation with the information from other representations. More specifically, this paper makes the following contributions:

- We study formalisms for representing ontology change (Section 2).
- We present a component-based framework for defining ontology change (Section 3).
- We present an ontology of basic change operations that provides the basis for inter-operations between various components in the framework (Section 4).
- We define complex ontology changes, which provide the basis for more efficient data transformation and advanced user-interaction capabilities (Section 4).
- We present rules and heuristics for identifying complex changes between ontology versions (Section 5).

Throughout this paper, we use the following example to illustrate our concepts and ideas. Suppose that we are developing an ontology of wines. In the first version (Figure 1a), there is a class Wine with two subclasses, Red wine and White wine. The hierarchy also includes some specific types of red and white wines. Figure 1b shows a later version of the same ontology fragment. Note the changes: we introduced a new subclass of Wine, Rosé wine; the classes that were previously subclasses of White wine—Cabernet blanc, White Zinfandel, and Vin gris—are now subclasses of the new class Rosé wine; we renamed the Riesling class to Weisser Riesling.

2 Formalisms for Representing Change

There is a number of ways in which we can represent change information. On the one end of the spectrum of representation forms, we may have very few details about changes from V_{old} to V_{new}. For instance, the two versions of the ontology may be all the information that we have. On the other end of the spectrum, we may have a complete and detailed representation of changes from V_{old} to V_{new}: both versions, a detailed log of changes, conceptual description of changes, metadata about them, and so on.

The following are some of the ways to represent change information for an ontology version V_{new} (In a particular environment we can have one or more of these elements in place):

- the old version of the ontology V_{old} (providing the *basis* for finding change information but no explicit change information)
- a log of changes applied to V_{old} that result in V_{new} (providing a record of the ontology-transition process)
- a structural diff between versions that describes differences between them (providing a declarative view of the ontology transition)
- a set of conceptual changes between versions (providing an explicit specification of conceptual relations between concepts in V_{old} and corresponding concepts in V_{new})
- a transformation set that describes a sufficient set of change operations for the transition from V_{old} to V_{new} (providing an operational view of the changes)

One of the easiest change representations to create, with the appropriate tool support, is a **change log** between versions. A change log records an exact sequence of changes that occurred when an ontology developer updated V_{old} to arrive at V_{new}. Many ontology-editing tools, such as Protégé [1], OntoEdit [11] and others, record changes that developers make. There are several detailed proposals for the information that logs should contain (e.g., versioning in KAON [10], Concordia [8]). For example, the evolution framework of KAON provides a number of "add", "set" and "delete" operations. The log contains a list of specific operations, such as "AddPropertyDomain" or "RemoveSubConcept" with references to the concepts or properties that they operate on.

Most logs of ontology changes are quite similar to the KAON format. They contain simple ontology changes, where the level of granularity at which changes are specified is close to a single user-interface operation. A log is that it provides a complete and unambiguous change specification at a very fine level of detail. Figure 2 shows a possible log of changes between versions from Figure 1.

```
Feb 25 13:36, user,
     changeName oldName=Riesling, newName=Rheinriesling
Feb 25 13:37, user,
     changeName oldName=Rheinriesling, newName=Weisser Riesling
Feb 25 13:37, user,
     addSuperclass child=Cabernet blanc, parent=Rosé wine
Feb 25 13:37, user,
     removeSuperclass child=Cabernet blanc, parent=White wine
```

Figure 2: A fragment of a log of changes that took place in the example in Figure 1.

```
changeName oldName=Riesling, newName=Weisser Riesling
addSuperclass child=Cabernet blanc, parent=Rosé wine
addSuperclass child=Vin gris, parent=Rosé wine
removeSuperclass child=Cabernet blanc, parent=White wine
removeSuperclass child=Vin gris, parent=White wine
```

Figure 4: A fragment of a transformation set for the example in Figure 1.

Change logs may not always be available however. In a dynamic and de-centralized environment such as the Semantic Web, we may have access only to the old and the new version of an ontology, but not to the record of the change. Furthermore, change logs are less useful in an environment where several editors update an ontology at the same time: interleaving the logs to find out the final effect of changes is a difficult task in itself. Therefore, there are a number of ways to represent change that relate V_{old} and V_{new} directly, without taking into account the specific sequence of changes that has actually taken place.

A **structural diff** [6] provides a map of correspondences between frames in V_{old} and V_{new}. For each from in V_{old}, it identifies whether or not there is a corresponding frame in V_{new} (its image) or whether a frame was deleted, or a new frame was added. Figure 3 shows a structural diff between ontology versions in Figure 1. The structural diff shows that the class Rosé wine was added, the class Riesling was renamed into Weisser Riesling, the class Cabernet blanc changed its superclass, and so on. PROMPTDIFF [6] is an example of a tool that uses heuristics to create a structural diff automatically. It uses persistent identifiers of the frames in different versions, or, if such identifiers are not present, structural relations between ontology elements.

A structural diff provides a *declarative* view of changes: it represents the mapping between versions but not the operations to get from one version to another.

A set of **conceptual changes** specifies the conceptual relation between frames *across* versions, that is, the relation between a frame in V_{old} and the image of that frame in V_{new}. In our example in Figure 1, after creating the class Rosé wine, we moved a number of classes that were previously subclasses of White wine to the Rosé wine subtree. In this case, a conceptual change could specify that the class White wine in V_{new} is a *subclass* of the class White wine in V_{old}. Similarly, it could specify that Riesling in V_{old} is *equivalent* to Weisser Riesling in V_{new}. Sometimes, when a consistent interpretation of already annotated datasets is essential, updates are intentionally specified as sets of conceptual changes. For example, the EMTREE thesaurus,[3] which is used by Elsevier to index scientific publications, specifies updates by defining that specific terms become subsumed by other terms, or that they became synonyms of other terms.

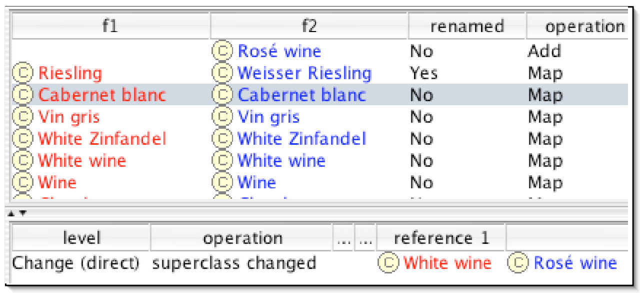
In the OntoView system [4], developers can augment a change description with conceptual relations between frames across versions.

A **transformation set** provides a set of change operations that specify how V_{old} can be transformed into V_{new}. Figure 4 presents one possible transformation set for versions in Figure 1. The transformation set in the figure contains only basic changes: each change is a single knowledge-base operation. The set can also include complex changes: for example, we can combine two operations that add a superclass and remove a superclass for the same class into a single move operation. We define transformation sets formally in Section 3 and we introduce basic and complex operations in Section 4.

A transformation set is different from a log in several aspects. First, while a log contains a record of *all* the operations that actually took place (including all intermediate steps) during the ontology-editing process, a transformation set specifies only the necessary operations to achieve the resulting change. Second, while a log is an *ordered* sequence of actions, there is only very limited partial ordering in a transformation set (mainly, that all "create" operations happen before all other operations). Third, while a log is a *unique* representation of the actual change process, there can be several (and often there are many) valid transformation sets for any two versions V_{old} and V_{new}.

Note that if a log of changes between two ontology versions consists of operations that do not undo other operations, this log is by definition a transformation set between these two versions.

The list of change representations in this section is not exhaustive. For example, some systems store concept-history information, associating with each concept a list of concepts that it was derived from, whether a concept was "retired" and which concept replaced it [8]. The systems with the primary purpose of data transformation may store a set of operations that is a specific "recipe" for transforming data instances. Other ways to represent change may develop as ontology evolution becomes more and more common.



f1	f2	renamed	operation
	Rosé wine	No	Add
Riesling	Weisser Riesling	Yes	Map
Cabernet blanc	Cabernet blanc	No	Map
Vin gris	Vin gris	No	Map
White Zinfandel	White Zinfandel	No	Map
White wine	White wine	No	Map
Wine	Wine	No	Map

level	operation	...	...	reference 1
Change (direct)	superclass changed		White wine	Rosé wine

Figure 3: A table representing a fragment of the structural diff between ontology versions in Figure 1 (generated by PROMPTDIFF).

[3]See http://www.elsevier.com/locate/emtree.

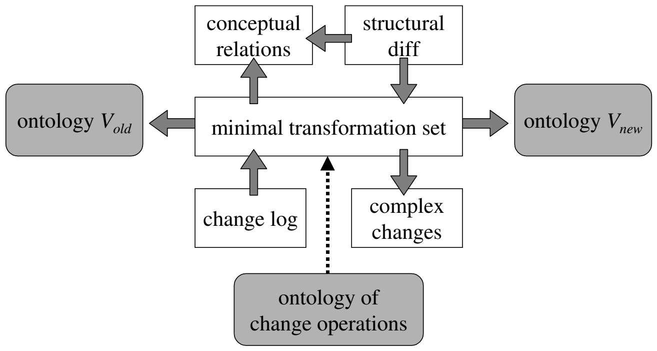


Figure 5: A schematic representation of the framework: a transformation set between two versions, specified with operations from the change ontology, and possible interactions with other change representations.

3 A Framework for Ontology Evolution

We now bring together the different formalisms that we described in Section 2 in a single ontology-evolution framework. In a distributed evolution environment, such as the Semantic Web, given an evolving ontology O, we can have *some* information about the change between two versions of O. For instance, we may have only the log of changes, or only a structural diff. However, once we have some of the change information, we can use additional tools to derive other information. For instance, we can use a log to derive a transformation set or we can use a structural diff to derive the definition of conceptual changes.

The purpose of our framework for ontology evolution is twofold:

1. relate the change information that is available in different formalisms (Section 2), and
2. provide mechanisms to derive new pieces of information from existing information.

Figure 5 shows the components of the ontology-evolution framework and some of the possible interactions between them.

The kernel of the framework is a *minimal transformation set*, which provides a set of operations that are necessary and sufficient to transform V_{old} into V_{new}. We already introduced the idea of a transformation set, here we define it formally.

Definition 1 (Transformation set) *Given two versions of an ontology O, V_{old} and V_{new}, a* ***transformation set*** *$T(V_{old}, V_{new})$ is a set of ontology-change operations that applied to V_{old} results in V_{new}. The operations in $T(V_{old}, V_{new})$ can be performed in any order, with one exception: all operations that* create *new classes, properties, and instances are performed first.*

A transformation set $T(V_{old}, V_{new})$ is ***minimal*** *if removing any operation from the set results in a set that is no longer a transformation set from V_{old} to V_{new}.*

A transformation set is not necessarily unique. The requirement that create operations are performed first has a very practical reason: some of the operations in the transformation set may refer to the newly created concepts. Thus they need to exist in order to be used in other operations.

We also define an *ontology of change operations* that can constitute a transformation set. This ontology is also a central element in our framework, because different tools using the framework must agree on the part of the ontology describing basic change operations. As tools use different formalisms for change representation for different tasks or augment information represented in one formalism with information in another, this set of basic operations is the "common language" that they share. This requirement to agree on a common set of basic change operations is similar to the requirement that agents on the Semantic Web share a common ontology language, such as OWL. Defining such an standard set is not unrealistic: once there is a common ontology language (e.g., once OWL becomes a standard), developing and agreeing on an ontology of basic changes is doable. Essentially, an ontology of basic changes is directly related to the ontology language itself and constitutes a set of simple operations to build an ontology in this language. We present the ontology of change operations in Section 4.

3.1 Interaction of Framework Components

As we mentioned earlier, we often have only an incomplete description of the change, with only some of the components in place. Different tools in the framework can use the available representations to derive new ones. As a result, having *some* information about change enables us to complete the picture by deriving additional elements of the change description. We use existing pieces of the puzzle to fill (some of) the missing pieces. Even in the case that we cannot fill in all the pieces, we might still be able to support tasks that we could not support before.

We now describe some of the transformations from one change description to another. Tools already exist for most of these transformations.

Change log → minimal transformation set Many ontology-editing tools provide logs of changes (e.g., Protégé, OntoEdit). These changes are often at the level of simple knowledge-base operations: adding a superclass to a class or removing one. We can transform logs into transformation sets by translating the operations into our vocabulary of basic changes (Section 4) and removing redundant changes.

Basic changes → complex changes If we have a transformation set consisting of basic operations, we can use heuristics to combine these simple operations to create complex change operations. For instance, if we have a set of siblings in a class hierarchy and each of these siblings had the same class added as a superclass and the original superclass removed, we can infer that the whole set of siblings was moved from one part of the hierarchy to another. In addition to the set of basic changes, we may need direct access to V_{old} to find complex changes. We describe such heuristics in Section 5.

V_{old} and V_{new} → structural diff If we do not have any specific information about the change, but we have both V_{old} and V_{new}, we can compare the two versions to create automatically a structural diff between them. For example, if concepts in an ontology have immutable concept ids, a simple tool can create a diff between versions identifying for each frame F in V_{old} its image in V_{new}. If we do not have immutable concept ids, tools such as PROMPTDIFF[6] use a set of heuristics based on concept names, class-tree structure, and concept definitions to create a structural diff.

Structural diff → transformation set If we have a structural diff, we can use it to create more useful change descriptions. Consider for example the structural diff in Figure 3. Knowing that Riesling became Weisser Riesling, we can add a changeName operation to the transformation set.

Transformation set → conceptual relations If we have a transformation set with both simple and complex operations defined between versions, we can use a set of heuristics to suggest conceptual relations between frames in versions to the user. For example, if we add a property to a class, we might suggest to the user that the new version of the class has become a subclass of the old version.

Structural diff → conceptual relations Similarly, we can use a structural diff to derive conceptual relations. For instance, the mappings of a structural diff directly suggest equivalence relations betweens concepts. In fact, we have integrated PROMPTDIFF, a tool that finds a structural diff [6], and OntoView, a tool for specifying conceptual relations [4], using the information produced by PROMPTDIFF to suggest initial conceptual relations in OntoView.

We outlined some of the ways to fill in missing pieces of the puzzle based on the pieces that we have. However, we cannot possibly envision what future tools will exist and what other ways to fill in new pieces researchers will come up with. Moreover, there could be more potential pieces of information that we may want to have, that our picture is currently missing. For example, several years ago, we might not have had either structural diffs or conceptual relations between versions in this picture. The main idea is to have this framework extensible. The guiding principle is "we'll take whatever input we have and do the best to find out more information."

3.2 Framework Components and Evolution Tasks

Each way to represent change in our framework is useful for some ontology evolution tasks that we listed in Section 1.

Minimal transformation set The minimal transformation set provides an operational view of the change. Thus we can use it for translation of other ontologies, for reversing changes, and as a starting point for data translation.

Complex changes Together with the minimal transformation set, we can use the complex operations to create data-transformation scripts. Also, complex changes allow us to determine in more detail the effect of changes on data accessibility and specific logical queries. Moreover, visualizing complex rather than basic changes, makes validation and approval tasks much easier for users.

Structural diff A structural diff is also useful for visualizing differences between ontologies. In addition, it is an essential information source for most of the supplementation techniques that are described in the next section.

Conceptual relations Finally, conceptual relations between concepts across versions facilitates data access by improving the interpretation and query of data sources that were described with different versions of ontologies.

4 Ontology of Change Operations

We have developed an ontology of change operations for the OWL knowledge model as an example of a common language for the interaction of tools and components in our framework.[4] The operations in this ontology are the elements for the specification of a *transformation set* (Definition 1). The ontology consists of two parts. The basis is an ontology of **basic change operations** and there is an extension that defines **complex change operations**. We chose the set of basic operations in such a way that the required commitment is minimal, while the set is still rich enough to capture enough knowledge about the change to derive new information.

4.1 Basic Change Operations

Each of the basic change operations modifies only *one* specific feature of the OWL knowledge model.[5] Examples of such operations are cardinality_change, changing the cardinality of a property restriction, or property_transitivity_addition, declaring a specific property as transitive. Altogether, we distinguish more than 80 different basic change operations.

We model change operations as a hierarchy of classes, where each class represents a specific type of change operation. The ontology specifies characteristics of each change type via property restrictions. All change classes use the `from` and `to` properties to refer to the source and target of the change.[6] In addition, most change classes have properties that specify an argument for a change operations. For example, one of the arguments for the cardinality_change operation is the integer specifying the new cardinality restriction.

By organizing the change operations in a class hierarchy, we exploit the inheritance mechanism to specify common properties of change operations in an efficient way. For example, all changes in property restrictions require two arguments to identify the source, namely the class identifier and the property identifier. Since all changes in property restrictions are subclasses of property_restriction_change, we can easily specify this fact for all operations at once.

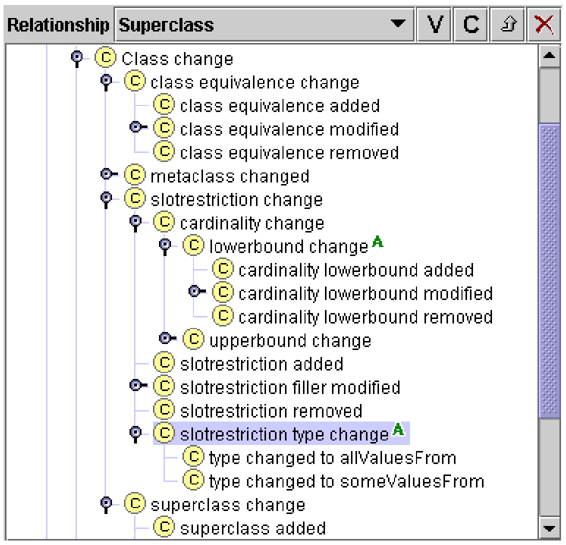
The ontology of basic changes contains "add" and "delete" operations for each feature of the OWL knowledge model. This set of operations ensures the completeness of the ontology of basic changes since it is sufficient for defining a transformation set from any ontology V_{old} to any other ontology V_{new}. While not being the most useful or efficient, such transformation set can contain the operations that delete all elements in V_{old} and then add all elements in V_{new}.

The ontology of basic changes also contains 'modify' operations, which specify that an old value is replaced by a new value. For example, a range_change operation specifies that

[4]A snapshot of a recent version of this ontology can be found at `http://ontoview.org/changes/1/3/`.

[5]See `http://www.w3.org/TR/owl-features/`.

[6]Additions and removals of classes or properties are the only exceptions, since they only have one of these two properties.


Relationship Superclass

- Class change
  - class equivalence change
    - class equivalence added
    - class equivalence modified
    - class equivalence removed
  - metaclass changed
  - slotrestriction change
    - cardinality change
      - lowerbound change
        - cardinality lowerbound added
        - cardinality lowerbound modified
        - cardinality lowerbound removed
      - upperbound change
    - slotrestriction added
    - slotrestriction filler modified
    - slotrestriction removed
    - slotrestriction type change
      - type changed to allValuesFrom
      - type changed to someValuesFrom
  - superclass change
    - superclass added

Figure 6: Some of the basic change operations.

the filler of the range of a property has changed. We can form these operations by combing a 'delete' and an 'add' operation. However, we have included them in the ontology of basic changes because this information will often be available. For instance, logs of changes provided by tools will often contain information on modifications. 'Modify' operation classes have properties for both old and new values, which give a reversible specification of the modification. Figure 6 shows a screenshot with some basic changes.

4.2 Complex Change Operations

In addition to the basic change operations, the ontology of change operations also contains complex change operations. Complex change operations are operations that are composed of multiple basic operations or that incorporate some additional knowledge about the change.

Complex operations thus provide a mechanism for grouping a number of basic operations that together constitute a logical entity. For example, a complex operation sib-lings_move consists of several changes of superclass relations.

Complex changes could also incorporate information about the implication of the operation on the logical model of the ontology. For example, a complex change might specify that the range of a property is *enlarged*, that is, that the filler of the range changed to a superclass of the original filler. To identify such changes, we need to query the logical theory of the ontology. In contrast, basic changes can be detected by analyzing the structure only.

We define the ontology of complex changes as an extension of the ontology of basic changes. We model specific variants of a basic change as subclasses of the basic change class. For example, the basic change superclass_change has two subclasses: superclass_changed_to_superclass and superclass_changed_to_subclass. Figure 7 shows a number of complex operations in our ontology.

Knowing complex operations rather than only the basic ones has a number of benefits.

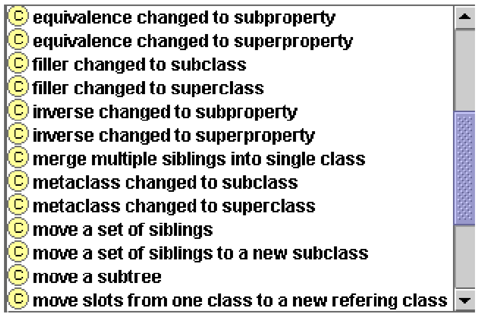

equivalence changed to subproperty
equivalence changed to superproperty
filler changed to subclass
filler changed to superclass
inverse changed to subproperty
inverse changed to superproperty
merge multiple siblings into single class
metaclass changed to subclass
metaclass changed to superclass
move a set of siblings
move a set of siblings to a new subclass
move a subtree
move slots from one class to a new refering class

Figure 7: A (non-hierarchical) list of some complex change operations.

- First, we can use complex operations to improve the user interface for the task of verifying and approving changes. Quite often, an ontology editor performs a number of changes that are all part of one "conceptual" operation. Some complex operations, like sibling_move, capture this knowledge. Visualizing these operations helps the user to verify modifications.
- Second, knowing complex operations, we can transform instance data with less data loss. For example, consider the move of a class: if we just had the "remove class" and "add class" operations, we will loose all instances of that class; knowing that the class was moved allows to move the instance data, too.
- Finally, knowing complex operations enables us to determine the effect of operations more precisely. If we only know that the range of a property has changed, we cannot tell anything about the effect on data. However, if we know that the range of the property is *enlarged*, we know that all old instance data is still valid.

If we know a set of basic change operations, we can use a set of rules and heuristics to *enrich* the basic changes with complex ones. We describe some of these *enrichment* procedures in Section 5.

The set of complex change operations is never finished or complete. It is always possible to define new complex changes that are useful in some setting. At the same time, a specific application does not have to use (or commit to) all complex change operations. The set of basic operations is already sufficient to specify all possible transformations.

Representing actual modifications By using an ontology to structure the operations, we do not need to define a syntax for the representation of actual changes. Instead, we can use the representation format that comes with the ontology language that we use. Our ontology of basic change operations uses RDF Schema as the "ontology language."[7]. Therefore actual changes can be represented as RDF data. Because both our ontology of changes and OWL itself can be represented

[7]We used RDF Schema instead of OWL because the expressivity of RDF Schema was sufficient to capture the main aspects of the ontology (the hierarchy of operations and their properties) In fact, because RDF Schema is a subset of OWL, we can also say that our ontology is represented in OWL

in RDF it is very simple to represent complex arguments. The RDF representation of the complex argument can just be inserted as value of a property in our ontology.

5 Finding Complex Operations

Existing tools that currently do provide change information, usually do it at the level of basic changes (e.g., logs of operations [10]). We are currently working on tools for identifying and presenting complex operations based on a set of basic operations or a structural diff between versions. In some cases, we can use a set of rules to generate a complex change from a set of basic changes (Section 5.1). For other changes, we may not have a definitive set of rules for finding the complex changes and will need to use heuristics to determine if a complex change occurred (Section 5.2). In the rest of this section we sketch some of the approaches for supplementing change description with complex operations that we are working on. However, as with everything else in the framework, these rules and heuristics can be extended with additional ones that other tools provide.

5.1 Using combination rules to find complex operations

Consider again the example in Figure 1. And consider the change to the classes that were subclasses of White wine in V_{old} and became subclasses of Rosé wine in V_{new}. We assume that we have an instantiated ontology of basic changes between the two versions. We can view this change as a set of several basic operations:

1. add a superclass relation between Rosé wine and Vin gris
2. remove a superclass relation between Vin gris and White wine
3. repeat the same for classes Cabernet blanc and White Ziinfandel

If we look at this set of operations conceptually, we can see that a complex operation was performed: a set of siblings was moved to a different location in the class hierarchy. Note that there are two levels of "enrichment" that we can identify in this example. First, we can recognize the "add superclass"–"remove superclass" sequence for each of the classes Vin gris, Cabernet blanc, and White Ziinfandel as a move in the tree. Second, we can recognize that Vin gris, Cabernet blanc, and White Ziinfandel were and remain siblings in the class hierarchy and thus we have a "move siblings" operation.

The set of rules to recognize this change is rather simple: as long as we know that the class A in the V_{new} is the same as the class A in V_{old} (this information is readily available in a structural diff) and their superclasses are different, we can identify a "move" operation. To recognize that a set of siblings was moved together, we compare arguments to the "move" operations. If the to and from arguments for a set of "move" operations are the same, we have a "move siblings" operation.

For some complex operations, just having a set of basic operations and access to V_{new} is not sufficient to determine that the complex operation has occurred. We may also need to have direct access to V_{old}. Suppose we know that a range of a property P was changed from C_1 to C_2. If we have access to V_{old}, we can check whether C_2 is a subclass of C_1 in V_{old}. If it is, then the range of the property P was restricted. As a result, for example, some instances that use this property may become invalid). On the other hand, if we know that C_2 is a *superclass* of C_1 in V_{old}, we also know that the range of the property P has become less restrictive (another complex operation). If our task is data transformation, we can conclude that no instances were affected. Therefore, if all we have, for example, are V_{new} and a transformation set with basic operations, these are the complex operations we will not be able to identify. Restricting a range of a property is an example of such operation.

5.2 Adding Uncertainty to Finding Complex Operations

While we can precisely identify when a group of sibling classes was moved to a new place in the class hierarchy, some other complex operations in practice may not have such precise definitions and may require a set of rules involving uncertainty to determine if the complex operation has occurred. In other words, while in principle we can specify a precise set of rules to determine when a complex operation occurs, in practice additional changes in the ontology involving the same concepts, may make a decision that a complex change has occurred less clear-cut.

Consider for example the following operation: group a set of siblings to create a new superclass (create a new abstraction). We would have such an example if we grouped the rosé wines in Figure 1a together to create the Rosé wine class, but have left this new class as a subclass of White wine.

In the ideal case, we have the following conditions that describe the case when such an operation has occurred:

1. A class $C \in V_{old}$ has n direct subclasses: $subC_1, subC_2, \cdots, subC_n$
2. There is a class $newC \in V_{new}$ such that:
   - $newC$ is a direct subclass of C
   - $\forall\, subC \in V_{new}$ such that $subC_i$ is a direct subclass of $newC$, $subC_i$ was a direct subclass of C in V_{old}

However, the user may have also, for example, added other subclasses to $newC$ (e.g., adding new types of rosé wines). He may have also added another level of classes between C and $newC$. Thus, we can rephrase the conditions for $newC$ above defining the heuristic to the following form:

- $newC$ is a subclass of C (not necessarily direct)
- Among the direct subclasses of $newC$, *most* come from $subC_1, subC_2, \cdots, subC_n$

We need to determine empirically what is a practical value for *most*. Our heuristic rule could say that more than 50% of subclasses of $newC$ must be former subclasses of C; or that there is at most one level of classes between C and $newC$.

Determining whether a class was split or two classes were merged into a single class has to rely on such heuristics even more unless an ontology-editing tool provides the merge and split operations directly and records them in the log. By just looking at two versions of an ontology and even at the transformation set including simple operations, it is often hard, if not impossible, to determine whether either of these operations has occurred. However, if we have some additional information, we may be able to identify such complex operation in some cases. In particular, if concepts have sets of instances associated with them or sets of properties that differ significantly, we can have the following set of heuristic rules (for merge in this example):

1. A class $C \in V_{old}$ has subclasses $subC_1$ and $subC_2$; I_1 and I_2 are sets of instances of $subC_1$ and $subC_2$; S_1 and S_2 are sets of properties for $subC_1$ and $subC_2$.
2. The class $C \in V_{new}$ has a subclass $subC$; I is a set of instances of $subC$; S is a set of properties for $subC$ and the following is true:
   - I is *similar to* $I_1 \bigcup I_2$
   - S is *similar to* $S_1 \bigcup S_2$

Here again we need to determine empirically the meaningful interpretations of *similar to*. Should the sets overlap by 90%? 50%? What if the classes were merged and then some of the instances of the merged class were deleted? Or new instances were added to the merged class?

We define the same set of rules and need to answer the same set of questions for the operations of splitting classes.

6 Summary And Outlook For Future Work

When we use ontologies in a distributed and dynamic environment, we need support for several ontology-evolution tasks, ranging from data translation to change visualization. Providing this support is difficult, as the distributed environment does not allow us to enforce specific development procedures, and we cannot count on having complete information about changes.

As a first step towards a solution of this problem, we have analyzed different formalisms for representing changes between ontologies and developed a framework that integrates and relates them. The framework specifies how we can supplement information in one representation with information from other representations.

An ontology of change operations is a key element in this framework. The ontology of change operations makes a distinction between basic operations and complex operations. Complex operations are often more useful for specification of the consequences of a change. We have described a number of rules and heuristics to distill complex operations from a set of basic operations.

The framework and the procedures that we described make it possible to supplement information about change, and thus to support ontology-evolution tasks that the original change information did not allow. However, the procedure does not guarantee, nor depend on, a complete description of a change. The more information we have, the more we can do with it.

Although many elements of the framework are already in place, a lot more work remains to be done. We need to experiment with the heuristics that we defined to test their effectiveness and to determine the optimal values for the parameters. We will need to build translators from existing change representations, such as change logs provided by specific tools to the vocabulary of basic changes. A crucial feature of the our framework is its extendability: it is always possible to define new interesting complex changes, new heuristics, or new and more complete ways of filling in missing change representations from available ones. Finally, we need change representations to build tools to support various evolution tasks. The more complete representation of change enables us to build better, more robust and more efficient tools.

References

[1] The Protégé project. http://protege.stanford.edu.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[3] J. Heflin and J. Hendler. Dynamic ontologies on the web. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA, 2000.

[4] M. Klein, A. Kiryakov, D. Ognyanov, and D. Fensel. Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Sigüenza, Spain, Oct. 1–4, 2002.

[5] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmr, and T. Risch. Edutella: a p2p networking infrastructure based on rdf. In *11th International Conference on World Wide Web*, pages 604–615. ACM Press, 2002.

[6] N. Noy and M. Musen. PROMPTDIFF: A fixed-point algorithm for comparing ontology versions. In *18th National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Canada, 2002.

[7] N. F. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003. in press.

[8] D. E. Oliver, Y. Shahar, E. H. Shortliffe, and M. A. Musen. Representation of change in controlled medical terminologies. *Artificial Intelligence in Medicine*, 15:53–76, 1999.

[9] H. S. Pinto and J. P. Martins. Evolving ontologies in distributed and dynamic settings. In *8th International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002.

[10] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Sigüenza, Spain, Oct. 1–4, 2002.

[11] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative ontology development for the Semantic Web. In *1st International Semantic Web Conference (ISWC 2002)*, volume 2342 of *LNCS*, pages 221–235. Springer, 2002.