Automating Instance Migration in Response to Ontology Evolution

Mark Fischer[1], Juergen Dingel[1], Maged Elaasar[2], Steven Shaw[3]

[1]Queen's University, {fischer,dingel}@cs.queensu.ca
[2]Carleton University, melaasar@gmail.com
[3]IBM, sshaw@ca.ibm.com

Abstract

Of great importance to the efficient use of an ontology is the ability to easily effect change [9]. This paper presents an approach toward automating a method for instance data to be kept up to date as an ontology evolves.

1 Introduction and Motivation

As computers become more ubiquitous and the information they attain and store becomes vast, the benefit gained from formally representing that information grows. Ontologies are one of the key technologies which strive to give information well-defined meaning. This, in turn, allows computers and people to work more cooperatively [10].

While there are many applications which help develop and create ontologies, there are still very few which aid or facilitate the evolution of an ontology [7]. Changes in domain understanding and changes in application requirements often necessitate a change in the underlying ontology [11]. It may be impractical or impossible to predict how or even if an ontology may change after being deployed. Changes to an ontology may generate inconsistencies in dependent ontologies. As ontologies evolve and grow, it becomes increasingly attractive to find efficient ways of keeping dependent information up to date.

A change in an ontology requiring dependent ontologies to be updated is a common problem. IBM's Design Manager is a collaborative design management software. It works with domains which are specified using ontologies to store, share, search and manage designs and models. The work presented in this paper was inspired, in part, by work IBM has done and problems IBM has encountered while developing Design Manager. Figure 1 helps describe the problem being addressed.

In Figure 1, there exists an original ontology (O), an evolved or updated ontology (O') and a set of dependent ontologies (I_1, I_2, ..., I_n). For all j, I_j

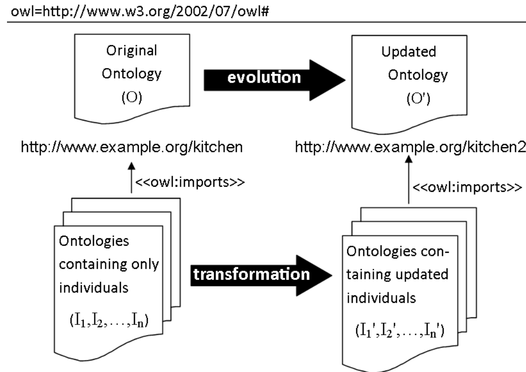

Figure 1: An overview of the problem setup

imports O and contains only individuals and facts asserted about those individuals.

Notice that classes and properties are kept separate from individuals. In an ontology, an individual is a resource which is an instance of a class and cannot contain any other resource (individuals may be related to one another through properties). Through this seperation, classes and properties are analogous to metamodels or schema while individuals are analogous to objects or data. Separating class definitions and property definitions from individuals is common practice when designing and building ontologies [3].

For any change that is made to O, there may exist an arbitrary number of j such that I_j is inconsistent. Furthermore, it is possible that there exist an a and a b such that a change in O will make I_a and I_b inconsistent for unrelated reasons (the axioms in O' that make I_a inconsistent are distinct from the axioms in O' that make I_b inconsistent). It is the hope that for all j, a transformation can co-evolve or migrate I_j into a new ontology, I'_j, such that I'_j imports O' and remains consistent. In Figure 1, the arrow labeled 'transformation' represents the portion of this problem for which we propose a unique solution which automates the transformation process.

Currently, this work is often done manually, or put off entirely because of the effort required to devise and implement these sorts of migrations. Due to the complexity possible in an ontology and the semantic ambiguity behind changes being made, it is not possible to automatically generate the transformation that performs the migration for individuals in dependent ontologies. Since ontologies may be too complex for a user to fully comprehend and changes may be ambiguous and therefore difficult for a computer or algorithm to deal with [11], the solution is to automate the migration process with the help of some user input.

To help automate the migration process, we aid the user creation of a transformation which unambiguously defines how to perform a migration for any possible dependent ontology. We also develop a set of tools and techniques

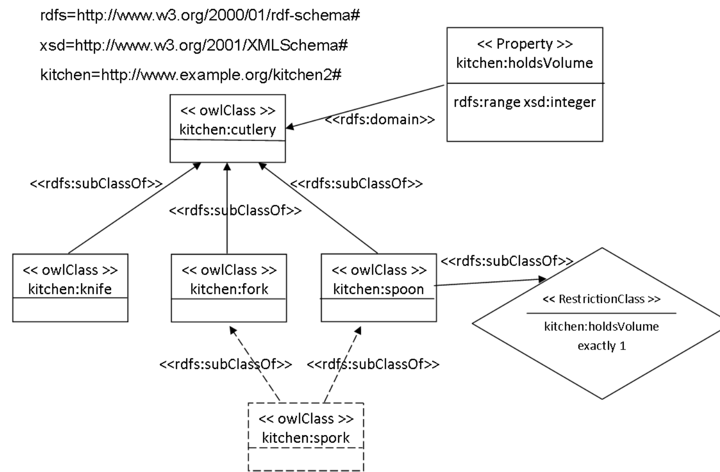

Figure 2: Kitchen example ontology and a possible evolution ontology. Solid components and their labels are from the original ontology while both solid and dashed components are from the evolved ontology. The original ontology has four named classes, one anonymous class, and a property. The evolved ontology has added a named class called *spork* which is a subclass of both *spoon* and *fork*. A spork is a pronged spoon and was first patented in the U.S. in 1874.

aimed at making it increasingly easier for users to create these transformations.

2 The Approach

The approach discussed here breaks the process into three phases which are incorporated into a single tool called Oital-T.

The first phase deals with comparing the two ontologies (O and O'). While not strictly necessary for the development of the transformation, an understanding of the difference between O and O' allows Oital-T to make suggestions and help the user to make choices regarding the transformation.

The second phase is the creation of the transformation itself. The transformation is authored in a language called Oital. Oital-T acts as an integrated development environment for Oital, allowing the syntax to be easier to learn and manipulate. The end goal of this entire process is to create a transformation written in Oital. This is the transformation that can then be run any number of times to migrate instances from O to O'.

Finally, the last phase involves analyzing the transformation. This phase of the process can be used to indicate which parts of the transformation may require correction or further development.

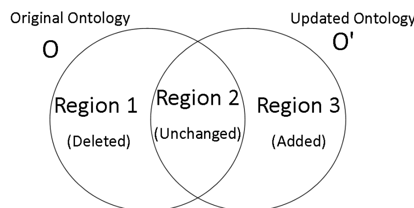

Figure 3: Venn Diagram showing regions where an axiom may be found

2.1 Analyzing the ontologies

In OWL, the separation of class and properties from individuals is expressed using axioms and facts. In relation to Figure 1, I_1 through I_n are all a series of facts with an inclusion reference to O while O and O' both consist of a series of axioms (possibly with inclusion references to other ontologies).

Axioms may either restrict or add to an ontology. Restrictive axioms add further clarification to the information encoded in an ontology (such as *holdsVolume exactly 1* in Figure 2) while additive axioms broaden the scope of the information (such as the creation of a new class).

Figure 3 depicts three regions in which an axiom may exist. Region 1 describes axioms which can be found in O and not in O' (deleted axioms), region 2 describes those axioms which can be found in both O and O' (unchanged axioms), and region 3 describes axioms which can not be found in O but can be found in O' (new or added axioms).

Axioms may influence one another. In Figure 2, the kitchen ontology has an axiom (A) stating that *cutlery* is the domain of the property *holdsVolume*. It also has the axiom (B) that there is a specific anonymous restriction class which contains only individuals that hold exactly one amount of volume. Axiom B is influenced by axiom A because a change in A is axiomatically a change in B as well. For any axiom, C, the axiom and all other axioms it is influenced by is denoted as C^* and called that axiom's context.

Table 1 gives an overview of how axioms are labeled while analyzing O and O'. If there exists axiom A such that its context, A^*, is entirely in Region 3, then A cannot be the cause of any inconsistencies. Specifically, if I_j is consistent with respect to O, then any inconsistencies I_j has with respect to O' cannot be due to axiom A. This means that aspects of the updated or evolved ontology which are entirely unique to the updated ontology are not of concern for the creation of a transformation. Similarly, for any A such that A^* is entirely in region 2, A cannot cause inconsistencies. This means that those parts of the original and updated ontologies which remain unchanged are not of concern either.

Axioms found in Region 1 may not, by default, be ignored. If the axiom is restrictive, such as the restriction class in Figure 2, then its removal cannot create an inconsistency and is considered OK. If, however, the axiom is additive, then its removal may create inconsistencies. For example, removing the *fork is a subclass of cutlery* axiom from the kitchen ontology in Figure 2 would result in

	Restrictive Axiom (and its context)	Additive Axiom (and its context)
Region 1	OK	Must Investigate
Region 2	OK	OK
Region 3	OK	OK
Region 1 & 2	Should Investigate	Must Investigate
Region 2 & 3	Must Investigate	Should Investigate

Table 1: Overview of which axioms require further investigation with respect to specific regions as outlined in Figure 3. *OK* means the axiom may be safely ignored as it is very unlikely to hinder a migration. *Should Investigate* means this axiom is unlikely to hinder migration, but it is probably semantically important. *Must Investigate* means this axiom needs to be considered carefully during migration as it may cause inconsistencies.

any facts which state that an individual fork holds a specific volume becoming inconsistent.

If, for a given axiom, A, the context, A^*, straddles two regions, then A is very likely to be semantically important. This means that even if the axiom may not cause inconsistencies, it may still be of interest to a user creating the transformation. Generally A^* having elements in region 1 and 2 suggests that a part of the ontology was removed but in such a way that the updated ontology had to be restructured to accommodate the removal (such as the removal of a class from a hierarchy). Similarly, A^* having elements in both region 2 and 3 suggests that a part of the ontology that has been added affects parts of the already existent ontology (Such as the creation of a new restriction class in a class hierarchy). Some of these changes may cause inconsistencies and are treated as more important, but all instances of these are shown to the user.

The following example illustrates how this analysis in performed. Consider the kitchen ontologies from Figure 2. The following is information that can be gathered from an analysis of these two ontologies. The *spork* class is new and therefore part of Region 3 from Figure 3. Its context includes the *fork* and *spoon* classes because of the subclass property. Because of the *spoon*, its context also involves the anonymous restriction class as well as the property *holdsVolume*. Because of *holdsVolume*, its context involves the *cutlery* class. Since *spork* is from Region 3, and it has a context which encompasses Region 2, this change straddles Region 2 & 3. The creation of a named class is additive, so (as shown in Table 1) this is unlikely to create an inconsistency during migration, but should still be brought to the user's attention as it may be important semantically.

2.2 Creating the transformation

A transformation that facilitates a migration of individuals from one ontology to another must, in some way, encode all the information that is still lacking after the two ontologies have been fully analyzed. While an analysis of the original

and evolved ontology can reveal how the original ontology has been changed, it cannot uncover the reasoning behind any given change.

A comparison of O and O' may uncover a difference for which there are many possible reasons. Consider the analysis of the evolution depicted in Figure 2, to know how to proceed, it is import to know why the *spork* class was created. If, for example, the class was added because of the discovery of a spork, then the migration can simply ignore this difference. If, on the other hand, the spork class was created because there were individuals within the class *fork* which held a volume, then a migration should take those forks which have the *holdsVolume* property and migrate them to the new *spork* class.

This very simple example illustrates why user interaction is required for the creation of the migration transformation. To express this transformation, we developed a domain specific transformation language called Oital.

2.2.1 Ontology Individual Transformation Authoring Language

Oital is a transformation language designed specifically for specifying the migration of individuals who are conformant to O such that they become conformant to O'. Currently, SPARQL update is used to perform tasks which alter an ontology. SPARQL update, however is an update language for RDF graphs. To use SPARQL update to effect change on an ontology requires an understanding of how ontologies are stored as RDF triples. Ontologies which are not stored in an RDF triple store must first be converted to one before SPARQL update can be used.

Oital alleviates these issues by using ontology concepts directly in the language (thus abstracting away from RDF). Instead of querying triples in a triple store the way SPARQL update does, Oital queries classes and properties in the ontology.

The current Oital compiler compiles Oital code into SPARQL update code. This means that ontologies queried using Oital must still be stored in RDF. It is, however, possible for Oital to be compiled in such a way that it can efficiently query and alter other formats as well. Details concerning the syntax and structure of Oital have been omitted due to space requirements.

To help make Oital easier to adopt, the syntax was heavily influenced by the Manchester OWL syntax. Oital transformation classes are defined using a syntax similar to the way OWL classes are defined. As the Manchester Owl Syntax has been adopted by World Wide Web Consortium [1], it should be fairly familiar to users who are already creating, updating, and evolving ontologies.

So far, no extensive evaluation of Oital has been conducted. However, preliminary results suggest that it is a convenient means of expression for the kind of ontology migration transformations we target.

2.3 Analyzing the transformation

Analyzing the created transformation is a helpful step toward inspecting if the transformation that has been created is correct. Currently, the only forms of

analysis supported by Oital's IDE (Oital-T) are traditional testing and a form of abstract interpretation which executes the transformation in some abstract fashion while collecting specific information [5][5].

We have developed and implements an abstract interpretation of Oital to track class membership. For instance, if after running an abstract interpretation of the transformation, the result shows that a specific class – such as *spoon* – is empty, then there exists no possible input such that any individuals from that input will have *rdf:type spoon*. Depending on the intent of the transformation, this may or may not be a desired result.

Traditional testing, abstract interpretation, and – in future – other forms of analysis may be used throughout the transformation authoring process in order to ease the development of complex transformations with fewer errors.

3 Related Work

The migration problem, as presented here, closely resembles the data migration problem found in database work [4] as well as the co-evolution problem found in model driven development (MDE) work [2].

Using a transformation language is an approach currently being used to solve the co-evolution problem. Languages such as ATL [6] (a QVT [8] compliant language) can be used to facilitate the automation of co-evolution in model-driven engineering [2]. When considering instance migration in ontologies, Oital is used in a similar capacity (in the domain of ontologies).

Not much work has been done on the migration problem for ontologies as has presented here, but similar work can be found concerning ontology evolution management. Much of the work done in this field, such as [11] relies on being able to access dependent ontologies (I_1, I_2, ..., I_n in Figure 1). They also depend on user involvement at the time of evolution or migration, which our approach does not require.

4 Conclusions and Future Work

In this paper, we have described an approach to facilitate the development of transformations that migrate individuals from the original ontology to an updated one. The approach is based on 1) differencing the ontologies, 2) transformation development using a novel, domain-specific language, and 3) analysis.

There is, however, still much outstanding work to be done. Along with continued development on Oital's IDE (Oital-T), we will attempt to identify transformation patterns and additional analysis that are scalable, yet still provide developers with useful information.

Aiding the development process will be sufficiently complex case-studies which shall help show the usability and capability of the approach presented in this paper. IBM has given us multiple versions of an ontology encoding of the UML specification along with a series of UML models stored as ontologies.

IBM has also offered access to a domain with IBM's Design Manager which may benefit from this approach.

References

[1] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language reference. February 2004.

[2] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, pages 222–231, 2008.

[3] A. Gangemi and V. Presutti. Ontology design patterns. *Handbook on Ontologies*, pages 221–243, 2009.

[4] J. Hall, J. Hartline, R. A. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2001.

[5] N. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. *Semantic Modeling, Clarendon Press, Handbook of Logic in Computer Science*, 4:527–635.

[6] F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Object-oriented Programming Systems, Languages, and Applications*. 21st ACM SIGPLAN Symposium, 2006.

[7] A. M. Khattak, Z. Pervez, S. Lee, and Y. Lee. After effects of ontology evolution. In *Future Information Technology*. IEEE 5th International Conference, 2010.

[8] I. Kurtev. State of the art of qvt: A model transformation language standard. *Applications of Graph Transformations with Industrial Relevance. Springer Berlin Heidelberg*, pages 377–393, 2008.

[9] N. F. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440.

[10] P. Plessers, O. De Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):39–49, 2007.

[11] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 285–300, 2002.