An Experimental Analysis of Tools for Ontology Evolution Management

Jéssica S. Santos[a], Viviane T. Silva, Leonardo G. Azevedo, Elton F. S. Soares and Raphael M. Thiago
IBM Research, Pasteur Ave, 146, Rio de Janeiro, Brazil

Keywords: Ontology Evolution, Ontology Engineering, Change Management, Change Detection, Complex Changes.

Abstract: The use of ontology is widely spread among software engineering groups as a way to represent, structure, share and reuse knowledge. As projects progress, the ontological understanding of the domain may change, evolve. New domain concepts may emerge and existing ones may disappear or be updated, causing changes in the ontology. Taking this into account, the management of the ontology evolution/life cycle should be addressed by the ones that adopt them as a way of representing knowledge. This paper provides an analysis of tools related to ontology evolution management focusing on the ones able to identify elementary and complex changes (by grouping elementary changes) that help on externalizing the intention of the user. The main contribution of this analysis is to present the state of the art of tools related to ontology evolution by identifying their strengths and limitations. The results are particularly useful to ontology designers who need to choose a tool to help them to inspect, understand and manage ontology evolution. Besides, we point out several research issues to be addressed in different kinds of research initiatives.

1 INTRODUCTION

In Computer Science, ontologies are defined as "explicit specifications of conceptualizations" (Gruber, 1993) or, similarly, as "formal specifications of a shared conceptualization" (Borst and Borst, 1997). Basically, they aim at providing a unified understanding of the world, formally representing the description of the concepts that exist in a given domain of interest, their relationships and relationships between instances that belong to such concepts.

Ontology usage presents several advantages, as for example: (i) interoperability – ontologies enable different systems or persons to communicate and interoperate by using a shared vocabulary of concepts (Maedche and Staab, 2001) that can be machine-processable (Fensel, 2001) (semantic exchange); (ii) reuse – when adopting ontologies, one can import and extend existing concepts, reusing knowledge (Uschold and Gruninger, 1996); and (iii) reasoning – the formally structured nature of ontologies allows logical reasoning and inference mechanisms to be exploited (Wang et al., 2004). Those advantages make ontologies a popular topic among the knowledge engineering community, which can adopt them in order to build/describe the structure of their knowledge base (Studer et al., 1998) and to construct Semantic Web based applications (Berners-Lee et al., 2001).

During the life cycle of an ontology, changes in its structure can be needed. This can occur, for example, when experts achieve a deeper understanding of the domain, when there are changes in the domain modelled by the ontology, when new requirements arise in the applications that the ontologies support (Zablith et al., 2015), or even when design errors are detected in earlier ontology versions (Hartung et al., 2012). Therefore, ontology-based applications are also subjected to a continual change (Abcckcr and Stojanovic, 2005).

In addition to elementary changes[1], there are others called composite or complex changes that group a set of elementary changes to provide more meaningful semantics. A *merge* operation is an example of complex change where a list of existing concepts $\{x_1,..,x_n\}$ are merged into one concept x_z that accumulates all attributes/relationships of the involved concepts. This change can be decomposed in several isolated changes without meaning: the creation of a new concept x_z, several move operations involving attributes/relations (from classes $\{x_1,..,x_n\}$ to x_z), and

[a] https://orcid.org/0000-0001-5082-4583

[1] Elementary changes are the ones that cannot be decomposed in simpler changes (e.g.: additions or deletions of concepts).

Santos, J., Silva, V., Azevedo, L., Soares, E. and Thiago, R.
An Experimental Analysis of Tools for Ontology Evolution Management.
DOI: 10.5220/0009395601110121
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 111-121
ISBN: 978-989-758-423-7

the sequence of deletions of classes $\{x_1, .., x_n\}$. The *renaming* is another example of complex change that can be decomposed into a sequence of changes without meaning: the deletion of a concept and the addition of a new one. However, these actions, when seen in isolation, provide the wrong semantics to the change (more examples of complex changes are presented in Table 2, marked by an asterisk (*)).

This research concerns with the ontology evolution management, the process that deals with supporting changes on the ontology, identifying and tracking them, and managing the propagation of changes to related artifacts (documentation, instances, code and dependent ontologies) (Stojanovic, 2004). This process is very challenging since when an ontology is modified, (i) ontology instances need to be changed to preserve consistency; (ii) dependent ontologies must be revised and immediately synchronized with the modified ontology; (iii) application code has to accommodate the changes on the ontology; and (iv) related documentations must be updated in order to document the evolution.

Inconsistencies between an ontology and its artifacts may cause severe problems, such as: (i) loss of application instances, *i.e.*, loss of knowledge generated by users when using the application; (ii) applications may behave unexpectedly; (iii) errors being propagated to other applications whose ontologies depend on the evolved one; and (iv) difficulty to maintain the system since the documentation is not up-to-date, *i.e.*, does not document the evolution. In this way, managing ontology evolution becomes more complex as the ontology grows in size (Klein and Fensel, 2001)(Stojanovic et al., 2002).

Some tools have been proposed in the literature to support or facilitate the ontology evolution management. In this research, we present an overview of existing tools and concentrate our experimental analysis on the ones that support features related to the detection/inspection of changes in a given ontology over time (diff tools[2]). One of the main features investigated in this research is the set of *kinds* of changes that can be captured and the level of explainability returned to the user as a conceptual interpretation of the meaning of a list of elementary changes (the so called complex change detection). We created a case study using the well known Pizza Ontology tutorial (Section 4), simulating several kinds of elementary and complex changes. These changes were selected based on our experience in designing ontologies, the changes presented in the tutorial, and complex changes presented by (Stojanovic, 2004), (Stojanovic et al., 2002) and (Klein, 2004).

[2]Tools able to track or detect ontology changes.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 presents an overview of tools related to ontology evolution. Section 4 details the evaluation criteria and the case study adopted to evaluate the tools. Section 5 presents a comparison between the tools and a discussion considering the supported features and the criteria established in Section 4. Lastly, Section 6 points out our conclusions and open issues identified.

2 RELATED WORK

The topic "ontology evolution" is the subject of several lines of research in the literature. Gonçalves (Gonçalves, 2014) argues that strategies for computing differences between ontologies can be classified as *syntactic* or *semantic*. Although his research is not focused on tools developed to cope with the ontology evolution management, some tools such as ontology editors that track changes are mentioned. Different from our work, Gonçalves (Gonçalves, 2014) does not explore the detailed list of specific kinds of changes that can be detected. Our research in this topic is much broader since we establish a set of criteria that are relevant to the ontology evolution problem and in addition to focus on evaluating diff tools, we present an overview of tools related to other parts of the ontology evolution process, as the ones that propose means to propagate changes.

The work presented in (Lambrix et al., 2016) focuses on the ontology visualization problem and discusses strategies that can be adopted to facilitate the visualization of ontology evolution. The authors argue that visualization tools must be able to represent the richness of ontologies in a human comprehensive way and present visualizations with different levels of granularity. Additionally, it emphasizes that existing techniques for the visualization of software evolution could be applied to address ontology evolution visualization. The strategy used to present differences between ontologies is only one of the features that we use to categorize tools in the present study.

The work in (Stojanovic, 2004) presents a brief discussion about ontology designing tools comparing them according to their ability to evolve an ontology. Different ontology editors were compared according to dimensions that include: functionality – the kinds of changes the ontology editor allows the user to apply on an ontology (*e.g.*, add, delete, copy concept); refinement – the ability to make recommendations to improve the ontology; reversibility – the ability to undo changes; usability – the ability to edit ontologies easily. In contrast, we do not concentrate our

research in ontology editors neither in their ability to implement changes, but in tools that detect changes between ontology versions (what may be done independently of a specific ontology editor).

3 ONTOLOGY EVOLUTION MANAGEMENT TOOLS

This section presents the tools that provide features for ontology evolution management. We present the most popular tools (based on citation numbers), namely, PromptDiff (Noy et al., 2002) and WebProtégé (Tudorache et al., 2013) and the ones that were designed to cope with specific features that are needed to understand ontology evolution (see Section 4 for more detail), namely, complex change detection (CODEX (Hartung et al., 2012) and KAON (Stojanovic et al., 2002)), change explanation (OWL Diff (Redmond and Noy, 2011)) and visualization (OntoDiffGraph (Lara et al., 2017)). In addition to diff tools, we also search for tools related to change propagation, however only one tool was found (OIM tool (Davidovsky et al., 2011)). A brief overview of these tools and their features will be presented as follows.

3.1 PromptDiff

The PromptDiff (Noy et al., 2002) is a tool that adopts a set of heuristic matchers for finding structural differences between two versions of an ontology. PromptDiff compares ontologies as follows (Noy et al., 2002):

- A matching between elements of the two ontologies is performed considering the elements' identifier (IRI) and type;
- When there is a matching between a class of the ontology original version and a class of the ontology new version and both classes have a single unmatched subclass, these subclasses are considered the same element (an entity renaming is identified);
- Unmatched siblings whose prefixes or suffixes changed in the same way in the new version of the ontology are considered the same (an entity renaming is identified);
- If there is a matching between a class of the original version of the ontology and a class of the new version of the ontology and both classes have a single unmatched slot[3] whose range is the same, they are considered the same element (an entity renaming is identified);
- If there is a matching between a slot of the ontology original version and a slot of the ontology new version and these slots have inverse slots that are unmatched, the unmatched inverse slots are considered the same elements (a slot renaming is identified);
- It can identify classes that were splitted or merged based on analysis of their instances;

In addition to the limited support for complex detection, another drawback of this tool is that it still uses an API for accessing OWL ontologies that is not able to support OWL 2[4] ontologies. Additionally, PromptDiff is only available as a plugin for Protégé[5] old versions (3.x and earlier series) and its source code is not available.

3.2 OWL Diff

The OWL Diff[6] (Redmond and Noy, 2011) is a tool that is able to compare different versions of the same ontology. It is available as a standalone application, and as a plugin for the Protégé (version 4.2 and higher). OWL Diff was inspired by PromptDiff and was developed to support OWL 2. The OWL Diff receives as input two OWL files and display the changes identified in a text log. Before identifying ontology structural changes, the following alignment strategies are considered (Redmond and Noy, 2011):

- Align by the same IRI and type;
- Align by rendering, *i.e.* concepts that have the same rendering (rdfs:label annotation property) are matched even if their IRI has changed;
- Align entities that differ only by their namespace;
- Siblings of a matched class are aligned when they have very similar rendering (it is interpreted as a refactoring to correct a typo or misspelling). It can also detect cases when all classes changed from plural to singular;
- Entities with the same parent and child are aligned;
- Alignment between alone unmatched classes when all the remainder classes are matched.

The support provided by this tool to complex detection is very limited since it only covers entity rename and subclass changes. Additionally, the heuristics presented are very simple and their alignments can point out wrong matches, as in the case of alignment of alone unmatched classes.

[3]Slots describe properties of classes or instances.

[4]https://www.w3.org/TR/owl2-overview/

[5]An open-source ontology editor

[6]https://protegewiki.stanford.edu/wiki/Protege4OWL- Diff

3.3 OntoDiffGraph

The OntoDiffGraph[7](Lara et al., 2017) is a tool that was developed to exhibit differences between ontologies in a graphical manner, by using graphs. The graph notation adopted is based on the VOWL (Lohmann et al., 2016) – a notation that maps each ontology element to a graphical element – but extends it in order to highlight differences. In summary, the graph notation is as follows. Classes are represented as blue circles, object properties are represented as blue rectangles, data properties are represented as green rectangles, annotation properties are represented as yellow rectangles, data types are represented as yellow rectangles with black borders and element characteristics (functional, inverse functional, transitive, symmetric, asymmetric, reflexive and irreflexive) are represented as white rectangles. The differences identified are displayed in different colors, according to the kind of change, by changing the color of the borders/edges of the elements. Additions, deletions and updates are displayed in blue, red and green colors, respectively. An example is illustrated in Figure 1. The strategy adopted to detect changes is based on the analysis of the IRI, which is the unique identifier of each element of the ontology. In this way, an addition is detected when there is a new IRI in the new version. Similarly, removals are detected when IRIs of the original version do not appear anymore in the new version. Updates are identified when the content of an existing element has changed in the new version of the ontology. This tool is not able to detect complex changes and the graphical approach to present the detected changes is not enough when the list of changes is too large.

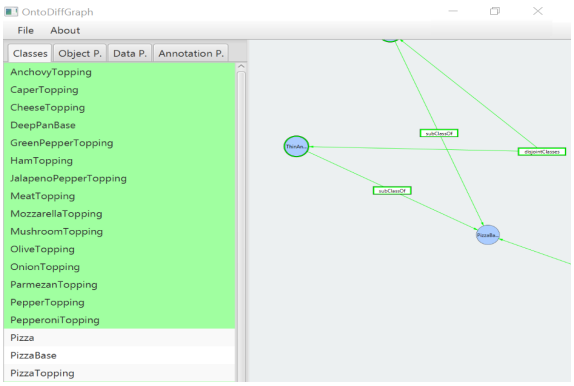

Figure 1: OntoDiffGraph Visualization.

[7]http://www4.di.uminho.pt/ gepl/OntoDiffGraph/

3.4 CODEX

The CODEX (Hartung et al., 2012) is a tool that was designed to find differences between life science ontologies. It supports as input ontologies described in the OBO [8] format, a representation language commonly used to describe life science ontologies. A limited support to OWL ontologies is also presented. First of all, the algorithm tries to find a matching between the elements of the different versions of the ontology. After that, it tries to identify elementary changes such as additions and deletions of concepts, relationships or attributes. Finally, a rule-based strategy is applied over the detected changes in order to find some kinds of complex changes, providing a compact result. The main algorithm to compute differences used by CODEX is called COntoDiff and is detailed in (Hartung et al., 2010). The CODEX mechanism to detect complex changes is able to detect: the addition or deletion of entire subgraphs, the addition or deletion of leaf concepts, the move of concepts, the split (tries to find a map between one concept of the old version and more than one concept of the new version) or merge of concepts (tries to find a map between two or more concepts of the old version and one concept of the new version), or even specific operations of life science ontologies (set the status of an active concept to be obsolete or the status of an obsolete concept to be active). Such an approach does not consider changes involving instances of the ontology since, according to the authors, life science ontologies typically do not have instances. The main drawback of this approach is that it cannot detect some kinds of simple changes (as the ones involving properties) due to its limited support to OWL. Additionally, the support to detect merge/split of concepts is also limited since it can detect such operations only in specific cases.

3.5 WebProtégé

The WebProtégé (Tudorache et al., 2013) is an application to support collaborative ontology management. This tool is available online and provides functionalities for editing, tracking changes and merging ontologies. Given the current version of the ontology loaded in WebProtégé *V1* and a version *V2* representing a set of changes externally applied to *V1*, one can merge *V1* and *V2* using WebProtégé in order to apply those changes described in *V2* to *V1*. Concepts that are in *V1* but are not in *V2* are interpreted as concepts removed. Since this tool records a change

[8]http://www.cs.man.ac.uk/h̃orrocks/obo/

history, the author of each change made inside the application is recorded. Before merging ontology versions, the changes detected between the versions are informed to the user and a confirmation dialog is exhibited. Additionally, it presents a functionality to revert/undo changes. The change history is presented in a text log and there is an option to filter changes according a given entity, facilitating change inspection. As a drawback, complex change detection is not well supported, covering few cases such as the detection of subclass changes. In addition, ontology elements cannot be renamed by using this tool, and renaming actions performed outside this editor are not detected too.

3.6 KAON

KAON (Stojanovic et al., 2002) is a framework that was developed to partially support the ontology evolution management. KAON does not support RDF but it adopts a proprietary ontology language that extends RDFS changing the semantics of domain, range and cardinality presented in the RDFS and OWL. This is because they are interpreted as constraints in the KAON language, and not as inference rules/axioms like in the OWL, RDFS. KAON provides some functionalities that allows the user to control and customize the process of ontology evolution, by setting some evolution strategies. When the ontology is modified, changes that will be propagated according to the predefined evolution strategies are also exhibited to the user. Examples of evolution strategies are as follows (Gabel et al., 2004):

- When a concept is deleted, the user can choose to delete orphaned concepts, reconnect them to the ontology root, or reconnect them to superconcepts;
- When a property is deleted, the user can choose to delete orphaned properties, reconnect them to superproperties, or leave them as they are;
- When concept's parent is removed, the user can choose to do not propagate properties or to add all inherit properties to the child concept.

Although the goal of this tool is to help the user while she is changing the ontology by indicating some changes propagation in the same ontology, it also records the changes made and, therefore, was also analyzed. Therefore, KAON provides a support to the implementation of changes in a given ontology. Among the limitations identified, we can cite that KAON uses a specific ontology language and does not present strategies for change detection or change propagation to other artifacts, relying on the fact that ontologies are only designed inside KAON.

3.7 OIM

The work in (Davidovsky et al., 2011) proposes a method for semi-automated updating ontology instances (ABox) accordingly to the TBox new version, process that the authors call *ontology instance migration*. The transformation rules needed to transfer instances of a given version to another one, are as follows: (i) concept addition – nothing happens to instances; (ii) concept rename – instances of the old name become instances of the new name; (iii) object property deletion – instances of the given relationship are discarded; (iv) object property addition – nothing happens to instances and is not possible to migrate instances automatically because the values of the respective object properties must be added to the instances but we do know the exact set of individuals that have to be related; (v) object property rename – instances of the relationship old name become the instances of the new name; (vi) object property cardinality change – some instances of the old name become the instances of the new name but is not possible to migrate instances automatically because we do not know the instances that should be related to the individual having the property whose cardinality changed; (vii) object property range change – nothing happens to instances. Similarly to cardinality change, instances cannot be automatically migrated; (viii) datatype property deletion – all instances of the relationship are discarded; (ix) datatype property addition – nothing happens to instances; (x) datatype property rename – all instances of the datatype property old name become instances of the datatype property new name; (xi) datatype property type change – all instances of the relationship old name are transformed to instances of the relationship new name having a different data type. The transformation rules of such an approach only exploits a small set of changes. Additionally, heuristics for change detection were not presented by the authors. It is important to note that OIM is the only tool analyzed that propagate changes to ontology instances. To the best of our knowledge no other tool is able to do such task neither is able to propagate ontology changes to code.

4 EVALUATION

This section presents the criteria used to evaluate the tools that provide features for ontology evolution management. Additionally, we describe the case

study used in the evaluation process.

4.1 Evaluation Criteria

The evaluation criteria were selected based on their importance to provide support to several parts of the process of ontology evolution, whose steps include the detection of ontology structural changes, the interpretation of their meaning, the analysis of their side-effects, and the propagation of changes to related artifacts. Since only one tool related to propagation of changes to ontology changes was found in the literature, we chose not evaluate this topic. The criteria are detailed as follows:

1. **Change Detection:** The elementary changes the tool is able to detect (listed in Table 1) and the *way* it is done. We identified two different ways in the analyzed tools: (i) store the ontology change history when the user uses the tool, providing the traceability between ontology versions; and, (ii) provide a mechanism to identify differences between two ontologies (*e.g.*, original ontology and its new version).
2. **Complex Change Detection:** Ability to identify complex changes, *i.e.*, to consider a composition of elementary changes as a unique change, *e.g.*, identify a concept renaming from a deletion and addition of concepts.
3. **Change Inspection:** How the detected changes are presented to the user. There are two main possibilities: (i) a list containing the changes is returned to the user as a log in text format; and, (ii) the list of changes is exhibited to the user graphically, *e.g.*, as a diagram.
4. **Extensibility:** Whether the tool can be extended, i.e., if it is provided as a plugin or even it its source code is available to be used.

The predefined criteria (1 – 4) are very relevant for the ones that adopt ontologies as a way of representing knowledge for the reasons explained as follows.

The existence of a list reflecting the modifications that an ontology underwent during its evolution – Criteria 1 and 2 – allows those that are not the authors of the changes to follow and understand how the ontology has evolved over time (Hartung et al., 2010), (Noy and Klein, 2003). Furthermore, the detailed information about ontology evolution is essential to investigate the real impact of the changes on related artifacts, and the manual detection of differences between ontology versions is a time-consuming task that can be very challenging for large ontologies (Kremen et al., 2011). The mapping between ontology versions is also very important for the ontology's migration. It consists in finding changes to transform an ontology *V1* in an ontology *V2* (Hartung et al., 2010). In this context, we categorize strategies for highlighting differences between versions as *change history* or *detecting differences* strategies. Note that, the need for specific tools for storing ontology change history occurs due to the fact that functionalities provided by traditional versioning systems (used to track source code changes like Git[9]) are not enough to track ontology changes. This is because they are based on *text comparison* (checking if two files have the same characters in the same order) and equivalent ontologies – ontologies that have the same axioms – can have different text representations, requiring a *structural comparison* (Noy et al., 2002).

The importance of detecting *complex changes* – Criterion 2 – instead of simply list elementary changes, occurs due to the fact that *complex changes* have a conceptual meaning and detecting them can help one to understand the user intention behind the performance of a list of elementary changes. Those changes represent composite/coarse-grained actions which can involve the combination of elementary changes such as a change in the concept hierarchy (Stojanovic, 2004). The ontology designer does not need to know, for example, the list of elementary changes indicating that a given concept *A* was detached from concept *B* and attached to concept *C*. He only needs to know that the hierarchy of concepts has changed. The explainability of a list of elementary changes (after an automatic interpretation) would be helpful to the ontology designers, leading to a proper propagation of changes and adjustments, making the evolution process faster, and facilitating the understanding of the ontology evolution. In some cases, to analyze each change of a list is not enough to capture the change semantics, as is the case of the rename action, explained as follows. A rename action changing the identifier of an element from *A* to *B* erroneously detected as the action of removing *A* and adding *B* can cause undesirable side-effects that would reflect in an erroneous evolution. This is because, when a class is removed, its instances also would be removed, what would lead to an instance loss for the renamed class *B* in the ontology newest version.

Criterion 3 was defined since a proper presentation of the set of detected changes can facilitate human comprehension (Lambrix et al., 2016).

Finally, Criterion 4 was defined since a partial feature support of a tool (or new feature needs) could be solved by extending the tool instead of building a new tool from scratch. We evaluated this criterion by checking tools documentation.

[9]https://git-scm.com/

4.2 Case Study

In order to evaluate the Criteria 1, 2, and 3 we simulate the ontology evolution process by applying a list of changes to a given ontology. We take as base a well known use case published in the literature, the Pizza Ontology[10] described in Protégé's tutorial[11], which describes in step-by-step exercises what the reader must do to create the Pizza Ontology. In summary, the Pizza Ontology describes the concepts of pizza, the country of origin of the pizza, toppings that can be added to the pizza, and so on. The first version of the ontology *V1* is an OWL file with no concepts. The next versions of the ontology are built by applying the changes described in the exercises (1 – 60) presented in the Pizza tutorial. Therefore, the ontology version *20*, for instance, is the one containing all changes made from exercise 1 to 20 and so on. Table 1 summarizes the complete list of changes considered in this research.

Some of the tutorial's exercises are not mentioned in Table 1 because they do not require the ontology to be changed (*e.g.*, "Exercise 30: Use the reasoner to automatically compute the subclasses of CheesyPizza"). The column *Protégé Tutorial Exercise* maps each kind of change to the exercise that implements it. Protégé's tutorial does not cover all possible changes. However, some changes can be derived from pre-existing ones by using a simple method: inverting the version order (*e.g.*, if the second version of the ontology is created by adding a class to its first version, the delete operation can be simulated taking the second ontology version as the original one and the first ontology version as the evolved ontology). Those cases are indicated with (-) in the column *Protégé Tutorial exercise*. Some kinds of changes were manually elaborated and included in our analysis and are represented with (x) in the column *Protégé Tutorial exercise*. The terms *object properties* and *data properties* refer to relationships between two individuals and relationships between individuals and data values, respectively. The term *restriction* refers to the class of individuals that contains a given property restriction (*e.g.*, class of individuals that only have vegetable toppings). The term *abstraction* refers to the inclusion or deletion of a class between parent/child classes. The property *characteristics* can describe object properties as being functional, inverse, transitive, reflexive, etc. Similarly, a data property can be characterized as functional. More information about property characteristics can be found in the Pizza tutorial.

[10]https://protege.stanford.edu/ontologies/pizza/pizza.owl

[11]http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorial P4_v1_3.pdf

Complex changes are signalled by an asterisk (*). The list of complex changes was built based on the combination of complex changes discussed in different approaches (Stojanovic, 2004), (Stojanovic et al., 2002) and (Klein, 2004).

5 COMPARATIVE ANALYSIS

In this section we present the result of the experimental analysis of different tools. In such analysis we evaluated the level of functionalities that each tool supports based on the criteria presented in Section 4. Among the tools surveyed, WebProtégé and KAON adopt the approach of ontology change history. Therefore, they do not need to discover what changes occurred in the ontology during its evolution because it is explicitly stored. Change history approaches have also the differential of keeping the exact order in which the sequence of changes were performed on the ontology, what it is not possible to know in approaches that detect differences without saving the trace. An advantage of using WebProtégé is that it also stores the author of each change performed inside it, *i.e.*, changes that were not incorporated by merging an external file. This factor is extremely important when the ontologies are managed collaboratively. The ability to revert changes also is another point in favour for both WebProtégé and KAON. A disadvantage of the KAON is that the management of the ontology is coupled with it, *i.e.*, the user will be able to inspect the detailed sequence of changes when the ontology is only edited using this tool. In contrast to change history approaches, tools that do not track changes but can detect differences between ontologies have the advantage that the ontologies to be compared can be edited using different ontology editors. This feature is specially desirable when the ontology designer reuses ontologies developed by independent organizations (Zablith et al., 2015).

Among the tools surveyed, the OntoDiffGraph was the only one designed to address the ontology evolution visualization problem, exhibiting the ontology as a graph, where different kinds of changes are displayed using different colors. Although this functionality can facilitate the visualization for the ontology engineer when the number of changes is not so numerous, for our case study it was difficult to visualize the ontology evolution since multiple changes occurred between versions. In order to deal with this problems, future tools for ontology evolution management could present means to collapse/expand parts of the ontology when adopting a graphical visu-

Table 1: Ontology changes.

	Kind of Change			Protégé Tutorial Exercise
1. Class	1.1. Include Class			7, 21, 22, 28, 31, 41, 43, 44
	1.2. Delete Class			-
	1.3. Change IRI*			x
	1.4. Annotation	1.4.1. Include Label		x
		1.4.2. Delete Label		-
		1.4.3. Change Label from A to B		x
		1.4.4. Include Comment		18
		1.4.5. Delete Comment		-
		1.4.6. Change Comment		x
	1.5. Description	1.5.1. Equivalent To	1.5.1.1. Include	60
			1.5.1.2. Delete	-
			1.5.1.3. Change	x
		1.5.2. SubClass Of	1.5.2.1. Include	6, 7, 18, 21, 22, 28, 31, 41, 43, 44
			1.5.2.2. Delete	-
			1.5.2.3. Change*	x
		1.5.3. Instances	1.5.3.1. Include	47, 58
			1.5.3.2. Delete	-
			1.5.3.3. Change	x
		1.5.4. Disjoint With	1.5.4.1. Include	5, 6, 23, 52
			1.5.4.2. Delete	-
			1.5.4.3. Change*	26, 27
	1.6. Convert to Defined*			29, 32, 41, 43, 54
	1.7. Convert to Primitive*			-
2. Object Property	2.1. Include property			8, 10
	2.2. Delete property			-
	2.3. Change IRI*			x
	2.4. Annotation	2.4.1. Include Label		x
		2.4.2. Delete Label		-
		2.4.3. Change Label from A to B		x
	2.5. Domain	2.5.1. Include Domain		14, 15
		2.5.2. Delete Domain		-
		2.5.3. Change Domain from A to B*		x
	2.6. Range	2.6.1. Include Range		13, 15
		2.6.2. Delete Range		-
		2.6.3. Change Range from A to B*		x
	2.7. Characteristics	2.7.1. Include		11, 12
		2.7.2. Delete		-
		2.7.3. Change		x
	2.8. Description	2.8.1. Inverse Property		10
		2.8.2. Disjoint Property		x
		2.8.3. SubProperty Of*		9
3. Data Property	3.1. Include property			46
	3.2. Delete property			-
	3.3. Change IRI*			x
	3.4. Annotation	3.4.1. Include Label		x
		3.4.2. Delete Label		-
		3.4.3. Change Label from A to B		x
		3.4.4. Include comment		x
		3.4.5. Delete comment		-
		3.4.6. Change comment from A to B		x
	3.5. Domain	3.5.1. Include Domain		47, 48
		3.5.2. Delete Domain		-
		3.5.3. Change Domain from A to B*		x
	3.6. Range	3.6.1. Include Range		47, 48
		3.6.2. Delete Range		-
		3.6.3. Change Range from A to B*		x
	3.7. Characteristics	3.7.1. Include		51
		3.7.2. Delete		-
		3.7.3. Change		x
4. Restrictions	4.1. Quantifier	4.1.1. existential restrictions (some)		16, 17, 19, 20, 21, 22, 28, 36, 37, 40, 41, 48, 49
		4.1.2. universal restrictions (only)		31, 34, 35
	4.2. Cardinality*			43, 45
	4.3. hasValue			59
5. Others* (changes involving more than one class or property)	5.1. SubTree	5.1.1. Include SubTree		7, 18
		5.1.2. Delete SubTree		-
		5.1.3. Move SubTree		x
	5.2. Abstraction	5.2.1. Include		x
		5.2.2. Delete		-
	5.3. Siblings	5.3.1. Include		4, 6, 22
		5.3.2. Delete		-
		5.3.3. Move		x
		5.3.4. Make all siblings disjoint		6, 7
	5.4. Merge Classes			x
	5.5. Split Class			x

* Complex Change.
x Kinds of changes that are not in the Protégé tutorial and were manually included.
- Kinds of changes that are not in the Protégé tutorial but can be derived from the existing ones.

alization.

In general, the support to complex detection is very limited. Only two approaches, namely, OWL Diff and PromptDiff address the entity renaming problem by presenting some simple heuristics that try to find renaming operations based on, for example, the analysis of the labels associated with the entities or based on the analysis on the parent and child of

the elements to be compared. Although WebProtégé stores the change history, it cannot detect IRIs renaming when ontologies are merged (loaded from an external file, compared and combined). We noticed that although subclass changes can be already well detected by some approaches such as CODEX, OWL Diff and PromptDiff, other complex changes such as splits and merges of concepts are not satisfactory identified since the approaches that deal with them rely on instance analysis and in many times instances of the ontology new version are not available. Furthermore, complex changes such as the conversion of a class to primitive or defined, changes involving siblings classes or changes creating/deleting abstractions are not addressed by any of the tools. In addition, domain/range changes are not properly interpreted by the tools (e.g.: they do not inform to the user that the range of a property has been enlarged when it is associated with a broader class). Although CODEX was designed to identify some cases of complex changes, it was not able to detect many cases of simple changes because it does not support all OWL constructs.

In the ontology evolution problem we want to identify differences between an old and a new version and, after that, automatically propagate changes to ontology instances based on the changes that were detected. A problem identified in relation to this issue is the lack of approaches supporting the automatic propagation of ontology structural changes to ontology instances (when it is possible), only the OIM tool proposes some rules to propagate changes to ontology instances. However, the rules considered by such an approach do not contemplate the complete list of changes presented in this paper. We also do not agree with some instance migration rules presented by the authors, such as the case in which the type of a data property changes. We argue that this case cannot be automatically solved since some conversions between types are not possible to be performed. KAON was the only tool that proposes a set of strategies that determine patterns for updating ontology elements when a change occurs.

Table 2 presents the changes that cannot be detected by the tools or that can be partially detected. All changes considered by OIM tool are described in section 3.7 by the transformation rules and are related to the following changes listed in Table 1: 1.1, 1.3, 2.1, 2.2, 2.3, 2.6.3, 3.1, 3.2, 3.3, and 3.6.3. Table 3 summarizes the characteristics of the surveyed tools. KAON and OIM tools are not mentioned in Table 2 because they do not propose own mechanisms to detect differences between ontology versions. The OIM tool is not mentioned in Table 3 because it does not support any of the criteria listed in the table. On the other hand, it is the only tool that has proposed automatic rules for automatic ontology instance migration, which is a factor that can be useful to the ontology evolution management.

Table 2: Change Detection Problems.

	Change	**No Support or Limited Support (*)**
Class	1.3. Change IRI	OWL Diff* OntoDiffGraph PromptDiff* WebProtégé CODEX
	1.5.1. Equivalent To	CODEX
	1.5.3. Instances	CODEX
	1.5.2.3. Change SubClassOf	OntoDiffGraph
	1.5.4. Disjoint With	CODEX
	1.5.4.3. Change Disjoint With	OntoDiffGraph, WebProtégé
	1.6. Convert to Defined	OWL Diff OntoDiffGraph PromptDiff WebProtégé CODEX
	1.7. Convert to Primitive	OWL Diff OntoDiffGraph PromptDiff WebProtégé CODEX
Object Property	2.3. Change IRI	OWL Diff* OntoDiffGraph PromptDiff WebProtégé
Data Property	3.3. Change IRI	OntoDiffGraph, PromptDiff, WebProtégé
Restrictions	4.2. Cardinality	CODEX
	4.3. hasValue	CODEX
Others (changes involving more than one class or property)	5.1. SubTree	OWL Diff OntoDiffGraph PromptDiff* WebProtégé
	5.2. Abstraction	OWL Diff OntoDiffGraph PromptDiff WebProtégé
	5.3. Siblings	OWL Diff OntoDiffGraph WebProtégé
	5.4. Merge Classes	OWL Diff OntoDiffGraph PromptDiff* WebProtégé
	5.5. Split Class	OWL Diff OntoDiffGraph PromptDiff* WebProtégé CODEX

* Limited support to detect this change.

Table 3: Tools characteristics.

	Change Detection		**Complex Changes**	**Change Inspection**		**Extensibility**
	Tracking changes	**Detecting differences**		**Text log**	**Graphic. visualiz.**	
OWL Diff		✓	✓*	✓		✓
OntoDiff-Graph		✓		✓	✓	
PromptDiff		✓	✓*	✓		
WebProtégé	✓	✓		✓		✓
KAON	✓			✓		
CODEX		✓	✓*	✓		

* Functionality partially supported.

6 CONCLUSIONS

In this paper we compared and evaluated different tools that present features related to the process of management of ontology evolution. We established a set of criteria to analyze the tools and employed a case study based on the Pizza Ontology to simulate the ontology evolution process. We applied different relevant kinds of changes in Pizza Ontology and analyzed how the tools deal with them.

By exploiting the features of existing diff tools we have concluded that none of the surveyed tools completely supports ontology evolution needs. A set of limitations have been identified, as follows:

1. **Change Detection:** The majority of tools are able to identify and return to the user/ontology designer a list containing the sequences of several elementary changes that occurred during the ontology life cycle. However, they deal with a different set of changes, and some of them (e.g.: CODEX) are not able to detect the complete list of elementary changes evaluated in this research (Table 2). In addition, most of tools analyzed (namely, OWLDiff, OntoDiffGraph, PromptDiff, WebProtégé and Codex) provide a mechanism to identify differences between two ontologies instead of tracking user changes.
2. **Complex Change Detection:** Although many diff tools have been found in the literature, they do not can deal with complex change detection properly since the identification of complex changes is still highly limited and partially addressed, as discussed in Section 5.
3. **Change Inspection:** Even tools designed to cope with visualization issues cannot present extensive list of changes in a comprehensive way.
4. **Extensibility:** Some tools tackle specific problems, but are not extensible (KAON, OntoDiffGraph, CODEX, PromptDiff) since they are not provided as plugins and their source code is not freely available.

In addition, we also notice that tools that propagate changes to impacted queries, dependent ontologies or source code have not been found. Only one tool (namely, OIM tool) proposes limited heuristics related to changes in instances.

The open problems found in the literature highlight the need for future work in the following directions:

- Improvement of existing techniques for complex change detection by performing, for instance, a deeper structural analysis of *relationships* and *attributes* of the ontology elements. This entails the improvement of change presentation since complex change detection includes grouping and interpret related changes that can be exhibited to the user in a compact way;
- Creation of a broader set of transformation rules to migrate instances considering the list of changes presented in this paper;
- Development of tools to propagate changes to other related artifacts (and not only ontology instances), such as documentation, related code and queries;
- Development of extensible tools that address both the difference detection task and the propagation of changes to related artifacts;
- The integration of existing approaches can also be investigated aiming at building a tool that covers different tasks of ontology evolution management or ontology life cycle.

Our lines for further research include: (i) to extend this work by conducting experiments with ontologies used in real world scenarios and conducting a study about other evaluation criteria that would be relevant for ontology evolution; (ii) to create a unified list (taxonomy) of complex changes found in the literature, and relating them by hierarchical relationships when it is possible. In addition, in that sense we want to analyze the kinds of complex changes that can be formalized and the ones that depend on human interpretation; and (iii) development of a tool focused on detecting complex changes, returning to the user high level messages about the ontology evolution instead of a long list of simple/elementary changes.

ACKNOWLEDGEMENT

This project was executed under the Brazilian National Petroleum Agency (ANP) R&D incentive regulatory framework.

REFERENCES

Abcckcr, A. and Stojanovic, L. (2005). Ontology evolution: Medline case study. In *Wirtschaftsinformatik 2005*, pages 1291–1308. Springer.

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

Borst, W. N. and Borst, W. (1997). Construction of engineering ontologies for knowledge sharing and reuse.

Davidovsky, M., Ermolayev, V., and Tolok, V. (2011). Instance migration between ontologies having structural differences. *International Journal on Artificial Intelligence Tools*, 20(06):1127–1156.

Fensel, D. (2001). Ontologies: Dynamic networks of formally represented meaning. *Vrije University: Amsterdam*.

Gabel, T., Sure, Y., and Voelker, J. (2004). D3. 1.1. a: Kaon–ontology management infrastructure. *SEKT informal deliverable*.

Gonçalves, J. R. (2014). *Impact analysis in description logic ontologies*. PhD thesis, The University of Manchester (United Kingdom).

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220.

Hartung, M., Groß, A., and Rahm, E. (2010). Rule-based generation of diff evolution mappings between ontology versions. *arXiv preprint arXiv:1010.0122*.

Hartung, M., Gross, A., and Rahm, E. (2012). Codex: exploration of semantic changes between ontology versions. *Bioinformatics*, 28(6):895–896.

Klein, M. C. and Fensel, D. (2001). Ontology versioning on the semantic web. In *SWWS*, pages 75–91.

Klein, M. C. A. (2004). *Change management for distributed ontologies*. PhD thesis, Vrije Universiteit Amsterdam.

Kremen, P., Smid, M., and Kouba, Z. (2011). Owldiff: A practical tool for comparison and merge of owl ontologies. In *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 229–233. IEEE.

Lambrix, P., Dragisic, Z., Ivanova, V., and Anslow, C. (2016). Visualization for ontology evolution. In *2nd International Workshop on Visualization and Interaction for Ontologies and Linked Data, Kobe, Japan, October 17, 2016*, pages 54–67. Rheinisch-Westfaelische Technische Hochschule Aachen University.

Lara, A., Henriques, P. R., and Gançarski, A. L. (2017). Visualization of ontology evolution using ontodiff-graph. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lohmann, S., Negru, S., Haag, F., and Ertl, T. (2016). Visualizing ontologies with vowl. *Semantic Web*, 7(4):399–419.

Maedche, A. and Staab, S. (2001). Ontology learning for the semantic web. *IEEE Intelligent systems*, 16(2):72–79.

Noy, N. F. and Klein, M. (2003). Tracking complex changes during ontology evolution. *ISWC-2003 Poster Proceedings, Sanibel Island, Florida*.

Noy, N. F., Musen, M. A., et al. (2002). PromptDiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI*, 2002:744–750.

Redmond, T. and Noy, N. (2011). Computing the changes between ontologies. In *Joint Workshop on Knowledge Evolution and Ontology Dynamics*, pages 1–14.

Stojanovic, L. (2004). Methods and tools for ontology evolution.

Stojanovic, L., Maedche, A., Motik, B., and Stojanovic, N. (2002). User-driven ontology evolution management. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 285–300. Springer.

Studer, R., Benjamins, V. R., Fensel, D., et al. (1998). Knowledge engineering: principles and methods. *Data and knowledge engineering*, 25(1):161–198.

Tudorache, T., Nyulas, C., Noy, N. F., and Musen, M. A. (2013). Webprotégé: A collaborative ontology editor and knowledge acquisition tool for the web. *Semantic web*, 4(1):89–99.

Uschold, M. and Gruninger, M. (1996). Ontologies: Principles, methods and applications. *The knowledge engineering review*, 11(2):93–136.

Wang, X., Zhang, D., Gu, T., Pung, H. K., et al. (2004). Ontology based context modeling and reasoning using owl. In *Percom workshops*, volume 18, page 22. Citeseer.

Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., and Sabou, M. (2015). Ontology evolution: a process-centric survey. *The knowledge engineering review*, 30(1):45–75.