EvoPat – Pattern-Based Evolution and Refactoring of RDF Knowledge Bases

Christoph Rieß, Norman Heino, Sebastian Tramp, and Sören Auer

AKSW, Institut für Informatik, Universität Leipzig, Pf 100920, 04009 Leipzig
{lastname}@informatik.uni-leipzig.de
http://aksw.org

Abstract. Facilitating the seamless evolution of RDF knowledge bases on the Semantic Web presents still a major challenge. In this work we devise *EvoPat* – a pattern-based approach for the evolution and refactoring of knowledge bases. The approach is based on the definition of *basic evolution patterns*, which are represented declaratively and can capture simple evolution and refactoring operations on both data and schema levels. For more advanced and domain-specific evolution and refactorings, several simple evolution patterns can be combined into a compound one. We performed a comprehensive survey of possible evolution patterns with a combinatorial analysis of all possible before/after combinations, resulting in an extensive catalog of usable evolution patterns. Our approach was implemented as an extension for the OntoWiki semantic collaboration platform and framework.

1 Introduction

The challenge of facilitating the smooth evolution of knowledge bases on the Semantic Web is still a major one. The importance of addressing this challenge is amplified by the shift towards employing agile knowledge engineering methodologies (such as Semantic Wikis), which particularly stress the evolutionary aspect of the knowledge engineering process.

The *EvoPat* approach is inspired by software refactoring. In software engineering, refactoring techniques are applied to improve software quality, to accommodate new requirements or to represent domain changes. The term refactoring refers to the process of making persistent and incremental changes to a system's internal structure without changing its observable behavior, yet improving the quality of its design and/or implementation [5]. Refactoring is based on two key concepts: *code smells* and *refactorings*. Code smells are an informal but still useful characterization of patterns of bad source code. Examples of code smells are "too long method" and "duplicate code". Refactorings are piecemeal transformations of source code which keep the semantics while removing (totally or partly) a code smell. For example, the "extract method" refactoring extracts a section of a "long method" into a new method and replaces it by a call to the new method, thus making the original method shorter (and clearer).

P.F. Patel-Schneider et al. (Eds.): ISWC 2010, Part I, LNCS 6496, pp. 647–662, 2010.

Compared to software source code refactoring, where refactorings have to be performed manually or with limited programmatic support, the situation in knowledge base evolution on the Semantic Web is slightly more advantageous. On the Semantic Web we have a unified data model, the RDF data model, which is the basis for both, data and ontologies. In this work we exploit the RDF data model by devising a pattern-based approach for the data evolution and ontology refactoring of RDF knowledge bases. The approach is based on the definition of *basic evolution patterns*, which are represented declaratively and can capture atomic evolution and refactoring operations on the data and schema levels. In essence, a basic evolution pattern consists of two main components: 1) a *SPARQL SELECT query template* for selecting objects, which will be changed and 2) a *SPARQL/Update query template*, which is executed for every returned result of the SELECT query. In order to accommodate more advanced and domain-specific data evolution and refactoring strategies, we define a compound evolution pattern as a linear combination of several simple ones.

To obtain a comprehensive catalog of evolution patterns, we performed a survey of possible evolution patterns with a combinatorial analysis of all possible before/after combinations. Starting with the basic constituents of a knowledge base (i. e. graphs, properties and classes), we consider all possible combinations of the elements potentially being affected by an evolution pattern and the prospective result after application of the evolution pattern. This analysis led to a comprehensive library of 24 basic and compound evolution patterns. The catalog is not meant to be exhaustive but covers the most common knowledge base evolution scenarios as confirmed by a series of interviews with domain experts and knowledge engineers. The EvoPat approach was implemented as an extension for the OntoWiki semantic collaboration platform and framework.

Compared to existing approaches for knowledge base evolution, our declarative, pattern-based approach has a number of advantages:

- EvoPat is a *unified method*, which works for both data evolution and ontology refactoring.
- The modularized, *declarative* definition of evolution patterns is relatively simple compared to an imperative description of evolution. It allows domain experts and knowledge engineers to amend the ontology structure and modify data with just a few clicks.
- Combined with our RDF representation of evolution patterns and their exposure on the Linked Data Web, EvoPat facilitates the development of an *evolution pattern ecosystem*, where patterns can be shared and reused on the Data Web.
- The declarative definition of bad smells and corresponding evolution patterns promotes the (semi-)automatic *improvement of information quality.*

This paper is structured as follows: We describe the evolution pattern concepts in Section 2 and survey possible evolution patterns in Section 3. We showcase our implementation in Section 4 while we present our work in the light of related approaches in Section 5 and conclude with an outlook on future work in Section 6.

2 Concepts

The EvoPat approach is based on the rationale of working as closely as possible with the RDF data model and the common ontology construction elements, i. e. classes, instances as well as datatype and object properties. With EvoPat we also aim at delegating bulk of the work during evolution processing to the underlying triple store. Hence, for the definition of evolution patterns we employ a combination of different SPARQL query templates. In order to ensure modularity and facilitate reusability of evolution patterns our definition of evolution patterns is twofold: *basic evolution patterns* accommodate atomic ontology evolution and data migration operations, while *compound evolution patterns* represent sequences of either basic or other compound evolution patterns in order to capture more complex and domain specific evolution scenarios. The application of a particular evolution pattern to a concrete knowledge base is performed with the help of the EvoPat *pattern execution algorithm*. In order to optimally assist a knowledge engineer we also define the concept of a *bad smell* in a knowledge base. We describe these individual EvoPat components in more detail in the remainder of this paper.

2.1 Evolution Pattern

Figure 1 describes the general composition of EvoPat evolution patterns. Bad smells (depicted in the lower left of Figure 1 have a number of basic or compound evolution patterns associated, which are triggered once a bad smell is traced. Basic and compound evolution patterns can be annotated with descriptive attributes, such as a label for the pattern, a textual description and other metadata such as the author of the pattern the creation date, revision etc.

Basic Evolution Pattern (BP). A basic evolution pattern consists of two main components: 1. a SPARQL SELECT query template for selecting objects, which will be changed and 2. a SPARQL/Update query template, which is executed for every returned result of the SELECT query. In addition, the placeholders contained in both query templates are typed in order to facilitate the classification and choreography of different evolution patterns. Please note, that in the following we will use the term variable for placeholders contained in SPARQL query templates. These should not be confused with variables contained in SPARQL graph patterns, which, however, do not play any particular role in this article. The following definition describes basic evolution patterns formally:

Definition 1 (Basic Evolution Pattern). *A basic evolution pattern is a tuple* (V, S, U)*, where* V *is a set of typed variables,* S *is a SPARQL query template with placeholders for the variables from* V*, and* U *is a SPARQL/Update query template with placeholders referring to a result set which is generated by the SPARQL query template* S*.*

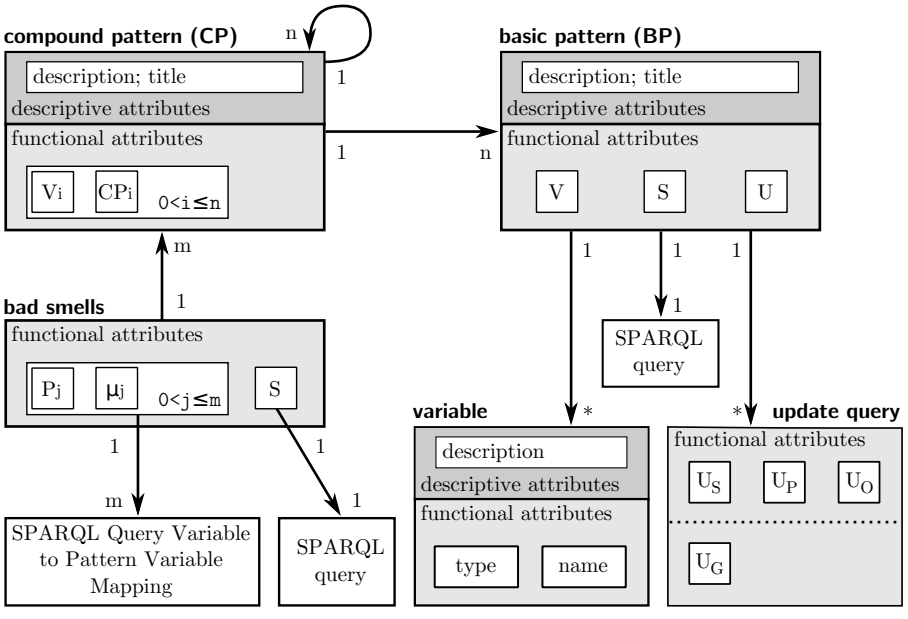

Fig. 1. Pattern composition with descriptive attributes, functional attributes and cardinality restrictions

```
1  V: dtProp type: PROPERTY
2     objProp type: PROPERTY
3     p type: TEMP
4     o type: TEMP
5  S: SELECT DISTINCT * WHERE {
6       %dtProp% %p% %o% .
7       FILTER (
8         !sameTerm(%p%, rdfs:range) &&
9         !sameTerm(%p%, rdf:type)
10      )
11    }
12 U: INSERT: %objProp% %p% %o% .
13    DELETE: %dtProp% %p% %o% .
```

Listing 1. Basic Evolution Pattern example: moving axioms from one property to another

Listing 1 shows a basic evolution pattern, which moves axioms from one property to another. Lines 1-4 define the typed variables used in the pattern. Lines 5-11 contain the SELECT query template, while lines 12-13 contain the SPARQL/Update query template to be executed for each result of the SELECT query.

Query preprocessor. In order to give a SPARQL query for previously unknown entities (since they are selected by the pattern SPARQL query), we introduce an extension to SPARQL that defines two additional types of variables and preprocessor functions:

- *Pattern variables* are enclosed in `%` characters and will be replaced with the corresponding entity. *Input variables* are defined by the user applying the pattern (e. g. on which entity the pattern is to operate.). *Temp variables* are variables to which query results from the pattern SPARQL query are bound. They can be used in the SPARQL/Update query of the same pattern to describe triple updates. In Listing 1, line 12 the variable `%objProp%` is used to bind the newly created object property.
- *Preprocessor functions* are a means of performing certain actions with the entities bound to a variable. If e. g. the user wants URIs of a certain format or change the datatype of a created literal value, those functions can be used. We provide a number of pre-defined functions for the most common use cases.

Compound Evolution Pattern (CP). Basic evolution patterns alone are not sufficient to cover arbitrary evolution scenarios. Especially on higher abstraction levels of represented domain knowledge, it is feasible to represent ontology changes on the same level of abstraction. To this end, we define compound evolution patterns, consisting of several evolution patterns that are subsequently applied to a knowledge base.

Definition 2 (Compound Evolution Pattern). *Let* $0 < i \leq n$, P_i *be (basic or compound) patterns and* V_i *the corresponding sets of unbound variables in* P_i. *A sequence* $CP := (V_i, P_i)$ *of patterns is called a* compound pattern *(CP).*

An example of a compound pattern for transforming a datatype property into an object property (including instance transformation) is given in listing 2. It consists of the following four basic sub patterns: moving property axioms, deleting datatype property, transforming instance data and creating object property.

```
1  // Sub pattern 1: (move axioms from dtProp to objProp)
2  V: dtProp type: PROPERTY
3     objProp type: PROPERTY
4     p type: TEMP
5     o type: TEMP
6  S: SELECT DISTINCT * WHERE {
7       %dtProp% %p% %o% .
8       FILTER (
9         !sameTerm(%p%,rdfs:range) &&
10        !sameTerm(%p%,rdf:type)
11      )
12    }
13 U: INSERT: %objProp% %p% %o% .
14    DELETE: %dtProp% %p% %o% .
```

```
15
16 // Sub pattern 2: (delete dtProp)
17 V: dtProp type: PROPERTY
18    p type: TEMP
19    o type: TEMP
20 S: SELECT DISTINCT * WHERE {
21      %dtProp% %p% %o% .
22    }
23 U: DELETE: %dtProp% %p% %o% .
24
25 // Sub pattern 3: (transform instance data)
26 V: dtProp type: PROPERTY
27    inst type: TEMP
28    o type: TEMP
29    objProp: PROPERTY
30 S: SELECT DISTINCT * WHERE {
31      %inst% %dtProp% %o% .
32    }
33 U: INSERT:
34    %inst% %objProp%getTempUri(getNamespace(%objProp%),%o%).
35    getTempUri(getNamespace(%objProp%),%o%) rdfs:label %o%.
36    DELETE: %inst% %dtProp% %o%
37
38 // Sub pattern 4: (create property)
39 V: objProp type: PROPERTY
40 S:
41 U: INSERT: %objProp% rdf:type owl:ObjectProperty .
```

Listing 2. Compound Evolution Pattern example: transforming a datatype into an object property while maintaining instance consistency

2.2 Evolution Pattern Processing

Algorithm 2.2 outlines the evolution pattern processing. The algorithm uses an evolution pattern P, a graph G and a set of variable bindings B as input. Depending on the type of pattern (basic or compound) the following steps are performed.

Basic pattern. If P is a basic pattern, the variables in the query are substituted with respect to their binding in B. Each of the update patterns contained in P is processed as follows:

1. If the update pattern sets an explicit graph, the active graph is set to that graph, else it is set to the default graph.
2. The variables in the update pattern are substituted according to B.
3. Changes are determined by executing the SPARQL query in P on G.
4. The changes are then applied to the active graph.

Compound pattern. Compound patterns are resolved to basic patterns. For each of the basic patterns the above steps are performed. The output of the algorithm is a set of changes on the respective graphs.

Algorithm 1. Pattern execution sequence

```
Require: Pattern P
Require: RDF graph G
Require: Variable bindings B
  if P is Basic Pattern then
    substitute variables in SPARQL Query according to B
    execute preprocessor functions in P
    QR := SPARQL query result of P on G
    for all update patterns of P as UP do
      if UP has graph then
        active graph AG = graph of UP
      else
        active graph AG = default graph G
      end if
      substitute variables in UP according to B
      generate changes CS of UP on AG with QR
      apply changes CS to AG
    end for
  else
    for all basic patterns in compound pattern P as SP do //maintain correct order
      execute Base Pattern SP //see above
    end for
  end if
```

2.3 Bad Smells

In order to assist knowledge engineers and domain experts as much as possible with the evolution of a knowledge base we also provide a formal definition for a bad smell in a certain knowledge base. In essence, a bad smell is represented via a SPARQL SELECT query, which detects a suspicious structure in a knowledge base. In most scenarios, there will be one (or multiple) evolution patterns addressing exactly the issue raised by a certain bad smell. Hence, we allow to assign one (or multiple) evolution patterns to the bad smell for resolving that issue. In order to further automatize the resolving of bad smells each evolution pattern can be assigned with a mapping from the bad smells result set to the variables used in the evolution patterns.

Definition 3 (Bad smell). *A bad smell is a tuple* $(S, (P_i, \mu_i))$*, where* S *is a SPARQL query and* (P_i, μ_i) *is a list of possible evolution patterns* P_i *for resolving the bad smell with an associated mapping* μ_i*, which maps results of* S *to the variables in* P_i*.*

```
SELECT ?s ?p ?o
WHERE {
  ?s ?p ?o .
  ?p a owl:DatatypeProperty .
  ?p rdfs:range ?range .
  FILTER (DATATYPE(?o) != ?range)
}
```

Listing 3. Bad smell example: selecting statements for which the datatype of the object doesn't match the `rdfs:range` of the property

An example of a bad smell is given in listing 3. It selects all statements whose object is a literal with a datatype that does not match the `rdfs:range` of the property of that statement. The result set from the bad smell query can be directly applied as input to a pattern that typecasts literal values to the correct datatype.

In certain cases a knowledge base evolution can be even performed completely automatically. This is the case if and only if both of the following conditions are met.

- The bad smell can only be resolved by exactly one evolution pattern and
- the mapping to the evolution pattern's variables is complete in the sense that all variables will be assigned values from the bad smell's query result set.

2.4 Serialization in RDF

To facilitate the exchange and reuse of previously defined evolution patterns we developed an RDF serialization, i. e. an RDF vocabulary for representing evolution patterns[1]. Together with an updated log publishing (such as e.g. proposed in [1]) on the Linked Data Web this facilitates the creation of an evolution ecosystem, where generic and domain specific evolution patterns are shared and reused and data cleansing and migration strategies can be also performed in network of linked knowledge bases.

3 Pattern Survey and Classification

In order to obtain a comprehensive catalog of evolution patterns we pursued a three-fold strategy: (1) we performed a comprehensive literature review, (2) we looked at all combinatorial combinations of before/after states and (3) we conducted a number of interviews with knowledge engineers and domain experts, which were involved in medium-scale knowledge base construction projects and retrospectively reviewed the evolution of these knowledge bases.

[1] The vocabulary for representing evolution patterns is available at: `http://ns.aksw.org/Evolution/`

Table 1. Combinatorially possible before/after evolution states. C, P, G stand for class, property, graph respectively. The '+' indicates that multiple entities of the same type participate in the evolution pattern. Impossible combinations are blackened out.

	$\emptyset$	$C+$	$P+$	$G+$	PC	PG	CG
$\emptyset$	ok	ok	ok	ok			
$C+$	ok	ok	ok				ok
$P+$	ok	ok	ok		ok	ok	
$G+$	ok			ok			
PC			ok				
PG			ok				
CG		ok					

Literature review. Most work concerned with ontology evolution patterns identifies a number of useful patterns but gives only an informal description which cannot be used for implementing an evolution software system. In [10], evolution patterns that work on the ontology level are identified. A classification of evolution patterns in four levels of abstraction is presented in [7]. The levels identified by the authors helped us in our classification system. In the interviews we conducted, the need for representational changes was identified. Thus, we added another layer that deals with syntactic changes to resources (i. e. renaming a URI). The authors of [3] present a number of patterns with formally defined participants and execution steps. We extended the approach, providing a pattern behavior in the form of SPARQL/Update queries that can directly be built into Semantic Web applications.

Combinatorial analysis. In order to ensure, that we achieved a comprehensive coverage of all possible evaluation patterns we followed a combinatorial analysis. We considered all possible combinations of ontology construction elements (i. e. classes, properties and (sub-)graphs) which are potentially affected by the application of a basic evolution pattern and the possible combinations of remaining elements after the pattern has been applied. All possible combinations are displayed in Table 1. For each of the potentially possible combinations we performed an analysis whether evolution patterns actually exist in practice. The results of this analysis are also summarized in Table 2. Combinations where possible patterns can be represented as combinations of basic evolution patterns are marked with a white background. Those combinations were no basic evolution patterns exist are blackened out.

Interviews and retrospective coverage checks. In order to ground our findings from the literature review and combinatorial analysis, we had an in-depth look at several medium- to large-scale knowledge base construction projects. These included in particular the Vakantieland e-tourism knowledge base for the Netherlands [9], the Leipzig Professors Catalog [2] and the development of an ontology for the energy sector, which was performed by our industry partner Business

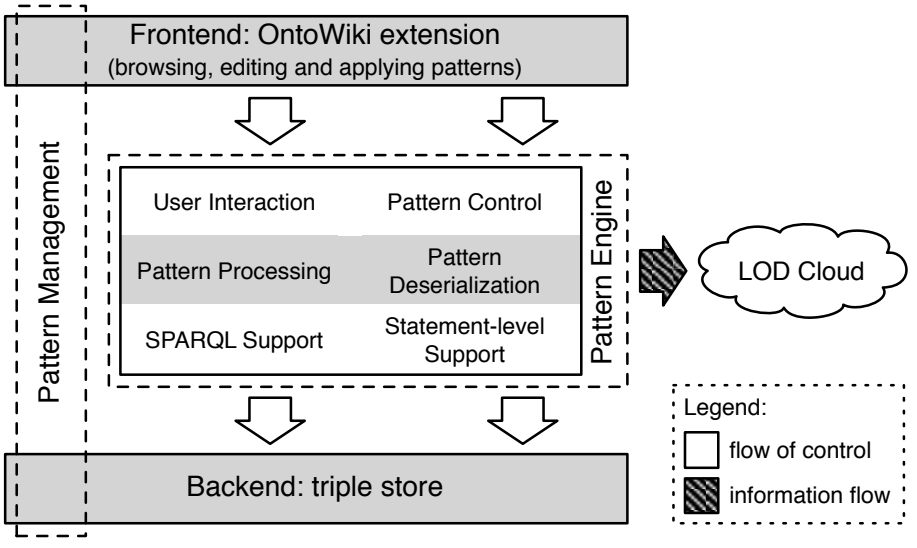

Fig. 2. System architecture with internal functional units and provided services. Patterns are exposed as Linked Data.

Intelligence GmbH. We also retrospectively reviewed the evolution of these knowledge bases and analyzed to what extend the previously defined evolution patterns would cover the found evolution steps.

4 Implementation

The EvoPat approach was implemented as an extension to OntoWiki – a tool for browsing and collaboratively editing RDF knowledge bases. It differs from other Semantic Wikis insofar as OntoWiki uses RDF as its natural data model instead of Wiki texts. Information in OntoWiki is always represented according to the RDF statement paradigm and can be browsed and edited by means of views. These views are generated automatically by employing ontology features such as class hierarchies or domain and range restrictions. OntoWiki adheres to the Wiki principles by striving to make the editing of information as simple as possible and by maintaining a comprehensive revision history. This history is also based on the RDF statement paradigm and allows to roll back prior change sets. OntoWiki has recently been extended to incorporate a number of Linked Data features, such as exposing all information stored in OntoWiki as Linked Data as well as retrieving background information from the Linked Data Web [6]. Apart from providing a comprehensive user interface, OntoWiki also contains a number of components for the rapid development of Semantic Web applications, such as the RDF API Erfurt[2], methods for authentication, access control, caching and various visualization components.

[2] `http://aksw.org/Projects/Erfurt/`

Table 2. Overview of valid evolution patterns on four levels of abstraction

Ontology level (OWL)		
Before	After	Description
$\emptyset$	$\emptyset$	Trivial empty pattern (no actions taken)
$\emptyset$	C+, P+ or G+	Creating class, property or graph
C+, P+ or G+	$\emptyset$	Deleting class, property or graph
C+	C+	Subclassing, union, merging, splitting classes
P+	P+	Property axioms (functional, symmetric, domain, range, etc.)
G+	G+	Graph merging and splitting, graph annotation
C+	P+	Remodeling from class membership to distinct property value
P+	C+	Remodeling from distinct property value to class membership
C+	CG	Class extraction from named graph
CG	C+	Merging classes into graph
P	PC	Converting datatype to object property
PC	P	Converting object to datatype property (incl. axioms)
Instance and data level (RDFS)		
Input	Output	Description
I^*	I	Instances merging
I^*, C^*	I^*	Instances reclassification
I^*, P, O	I^*	Adding data to instances
I^*, P, P^*	I^*	Generating data from existing instances data
I^*, L^*	I^*	Converting literal property values to resources
I^*, R^*	I^*	Converting resources to literal property values
$I^*(, P^*, O^*)$	I^*	Moving data (predicates and objects) from one instance to another
Entity level (RDF)		
Input	Function	Description
Literal, datatype	Setting datatype on literal	Datatype added, changed or removed
Literal, language	Setting language on literal	Literal language added, changed or removed
RegExp search/ replace	regexp replace	Performs a regular expression search and replace on literal value
Syntactic/representational level (RDF/XML, N3, etc.)		
Input	Function	Description
URI, namespace	Set URI prefix	Changes prefixes for a resource
URI, local name	Set local name	Changes local name of a resource

The general architecture of the EvoPat extension is depicted in Figure 2. It consists of four distinct components. Core of the EvoPat implementation is the *pattern engine*, which in particular handles processing, storing, versioning and exposing evolution patterns as Linked Data on the data web. It interacts via SPARQL with a triple store representing the EvoPat *backend*. The EvoPat *frontend* facilitates the user friendly browsing/selection, configuration and application of evolution patterns. The *pattern management component* as a logical component spans several architectural layers. It implements the required APIs needed by the user interface and backend for managing patterns.

Different versions of ontologies resulting from applying evolution patterns can be managed through OntoWiki's versioning component. Similar to database transactions, the changes on the statement level that result from applying a certain evolution pattern can be grouped and versioned as a single change.

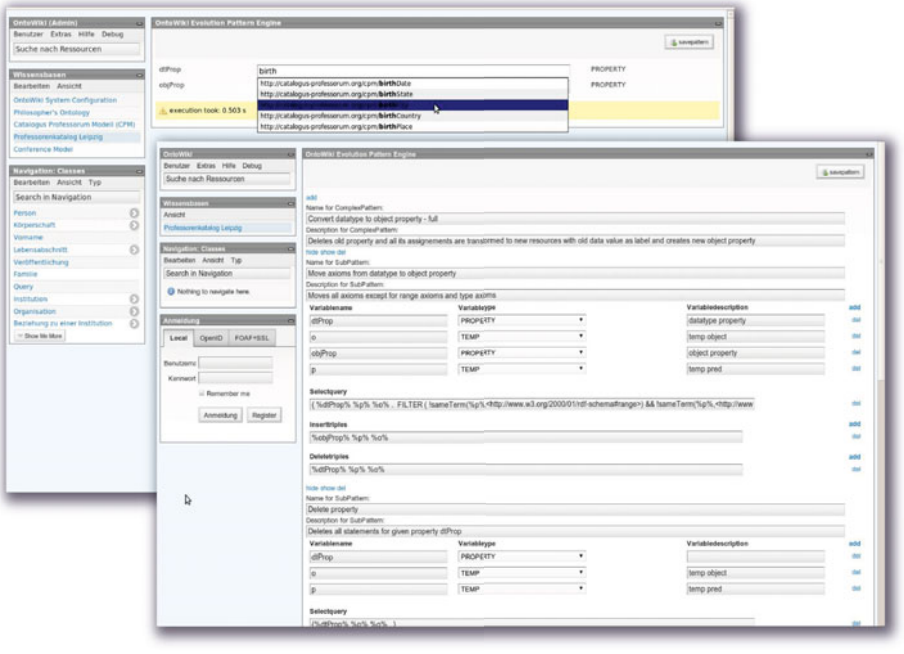

Fig. 3. EvoPat user interface showing pattern editor (right) and pattern execution view (left)

Figure 3 showcases the EvoPat user interface with the pattern editor and the pattern execution. The pattern editor allows to create basic and compound evolution patterns. A user friendly form is generated, where the descriptive attributes, the variables used in the pattern and the respective SPARQL SELECT

and UPDATE queries can be filled in. For pattern execution (as shown in the upper left part of Figure 3), the EvoPat implementation generates a form based on the variables definition of the evolution pattern at hand. Employing the typing of the variable a type ahead search simplifies the selection of concrete values for the variables.

Scalability evaluation. One of the main goals of developing EvoPat was to push as much of the evolution pattern processing down to the triple store. In order to evaluate whether EvoPat lives up to this promise we evaluated the processing of selected evolution patterns with different knowledge base sizes. The results of the evaluation are summarized in Table 3. We used the Catalogus Professorum Lipiensis knowledge base and simply created three different versions of it in different sizes, by simply copying the data. The results of the performance evaluation show, that the evolution pattern processing grows linearly with the knowledge base size. As a consequence, EvoPat can be used with arbitrarily large knowledge bases, the performance of the evolution pattern processing primarily depends on the speed of the underlying triple store.

Table 3. Scalability evaluation with two compound patterns on Catalogus Professorum Lipsiensis. The benchmarks were performed in three different sizes of the original knowledge base: original size (150K triples), 3 × the size (450K triples), 5 × the size (750K triples). Figures are quoted for two patterns each KB size.

	pattern exec. [s]	affect. rsrc. [pcs]	throughput [$\frac{\text{pcs}}{\text{s}}$]
KB size: 1 × 150K triples			
Datatype to Object Property	8.593	1300	151.3
Class merging	5.949	1500	252.1
KB size: 3 × 150K triples			
Datatype to Object Property	24.813	3900	157.2
Class merging	17.753	4500	253.4
KB size: 5 × 150K triples			
Datatype to Object Property	39.822	6500	163.2
Class merging	30.603	7500	245.1

5 Related Work

Ontology evolution has constantly been under research during the past two decades. In recent years a ramp-up could be observed due to Semantic Web research activity, thus providing a more user-centric view on ontology evolution.

A comprehensive overview on the field of ontology change is given in [4]. The authors conduct an extensive literature review, extracting and defining common

vocabulary as a base for discussion. They define ontology evolution as a "response to a change in the domain or conceptualization". The term ontology evolution, as used in this paper, covers what Flouris et al. refer to as ontology translation and by which they mean changes in the syntactical representation of the ontology (e. g. changing the URI of a resource).

To the best of our knowledge, there is no existing approach for formally specifying modular evolution patterns in a declarative manner. The most closely related approach in this regard is a categorization of pattern-based change operators in [7]. The paper defines four levels of abstraction of an ontology (element, element context, domain-specific and generic abstract level) to whose elements the said operators can be applied. Taking into account the Semantic Web infrastructure, our approach defines an additional level on the representation layer.

Stojanovic et al. in [12] define three requirements for ontology evolution: 1) ensuring consistency, 2) allowing the user supervision of evolution and 3) advice for continuous ontology refinement. In addition, the authors identify six phases of ontology evolution, namely 1) capturing, 2) representation, 3) semantics of change, 4) implementation, 5) propagation and 6) validation of changes. The KAON API[3], implementing the approach, also introduced by the authors. Furthermore, they identify the need for representing changes on different levels of granularity. To cope with different methods of applying changes to an ontology, they introduce basic evolution strategies, which define the steps of a complex evolution process. For a given change request there are usually more than on applicable strategy, resulting in different ontologies. Seen in a broader sense, these basic evolution strategies can be combined into so called advanced evolution strategies, of which they introduce four. Our compound patterns are similar in nature to Stojanovic's basic evolution strategies, but differ in the inclusion of explicit declarative semantics by means of SPARQL/Update queries.

An interesting approach to ontology evolution with particular respect to consistency management is given by Djedidi and Aufaure [3]. They propose a process model, an attached pattern and a versioning layer. If applying a change pattern results in a match to an inconsistency pattern, an alternative pattern is automatically applied by the proposed system. Furthermore, a quality assessment step is integrated into the process. The system can thus alleviate the need for user interaction by applying quality-improving patterns in an automated fashion.

Noy and Klein determine in [10] to what extent ontology evolution resembles schema evolution, which has been extensively researched in the database community. By arguing that different versions of an ontology have to be kept in parallel, they conclude that the traditional distinction between schema evolution and schema versioning is not applicable to ontology evolution and ontology versioning. Even though, EvoPat distinguishes between versioning and evolution, both subsystems are closely related and cannot be used exclusively. All evolutionary changes are automatically versioned and can be reverted at any time.

[3] `http://kaon.semanticweb.org/developers`

A declarative update language for RDF graphs, named RUL is defined in [8]. RUL is based on RQL and RVL and ensures consistency on the RDF and RDFS levels. It, therefore, contains *primitive*, *set-oriented* and *complex* updates as compositions of primitive or complex ones. Primitive RUL updates are similar in expressiveness to SPARQL 1.1 updates. Complex updates are expressed by means of fine-grained updates on class and property instance level. Our basic evolution patterns with variable placeholders are similar to the set-oriented RUL updates (i. e. repeating the same query for several bindings). Additionally, we, however, define a functional extension that allows for arbitrarily replacing entities in a preprocessor-like manner.

Finally, applying the software engineering concept of *code smell* [5] to ontologies has been inspired by the work of Rosenfeld et al. [11]. They use *bad smells* in a Semantic Wiki context for triggering refactoring operations.

6 Conclusion and Future Work

We introduced an approach to pattern-based evolution of RDF knowledge bases. By considering the complete stack of Semantic Web knowledge representation techniques including its syntactic infrastructure as opposed to just the ontology layer, our approach fulfills additional requirements identified for example in user interviews (cf. Section 3). We provide a concrete implementation that leverages the plug-in architecture of OntoWiki[4], our semantic collaboration platform and framework. Thus, our implementation can make use of existing functionality of the OntoWiki framework like versioning of RDF knowledge bases.

Currently, EvoPat only ensures consistency through the definition of consistency-preserving patterns by the knowledge engineer. User-defined patterns can, however, lead to inconsistent knowledge bases. An approach that ensures consistency by proposing only those patterns whose application will not result in an inconsistent ontology, would thus be desirable. A straightforward (but admittedly not very scalable) solution to this problem is to combined EvoPat with a reasoner and test the application of a pattern employing the reasoner before its actual application in order to ensure correctness.

As opposed to bad smells, which indicate modeling problems, a promising approach is also to share and reuse modeling best practices. A problem which has to be solved in this regard, is the formalization and elicitation of a user's modeling requirements. A related idea for future work is the consumption of Linked Data. Our current implementation publishes evolution patterns on the Data Web but makes no use of gathering further information about resources. Doing so, could deliver hints for the applicability of specific patterns.

In a number of application projects we learned, that a key factor for the success of a knowledge engineering project is the efficient co-design of knowledge-bases and knowledge-based applications. Through the declarative definition of evolution with EvoPat it becomes possible to (semi-)automatize this co-design, since a knowledge base refactoring can trigger code refactoring and vice versa.

[4] Online at `http://code.google.com/p/ontowiki/wiki/ExtensionCookbook`

References

1. Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., Aumueller, D.: Triplify: light-weight linked data publication from relational databases. In: Quemada, J., León, G., Maarek, Y.S., Nejdl, W. (eds.) Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, pp. 621–630. ACM, New York (2009)
2. Augustin, C., Kuchta, B., Morgenstern, U., Riechert, T.: Datenbank und website catalogus professorum lipsiensis. ein sozialstatistisches analyseinstrumentarium und seine repräsentation im netz. In: Schattkowsky, M., Metasch, F. (eds.) Biografische Lexika im Internet. Bausteine, vol. 14, pp. 167–184. TUDPress, Verlag der Wissenschaften GmbH, Dresden (2009)
3. Djedidi, R., Aufaure, M.-A.: ONTO-EVO^{A}L an Ontology Evolution Approach Guided by Pattern Modeling and Quality Evaluation. In: Link, S., Prade, H. (eds.) FoIKS 2010. LNCS, vol. 5956, pp. 286–305. Springer, Heidelberg (2010)
4. Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G.: Ontology change: classification and survey. Knowledge Eng. Review 23(2), 117–152 (2008)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
6. Heino, N., Dietzold, S., Martin, M., Auer, S.: Developing Semantic Web Applications with the OntoWiki Framework. In: Networked Knowledge – Networked Media. Springer, Heidelberg (2009)
7. Javed, M., Abgaz, Y.M., Pahl, C.: A Pattern-Based Framework of Change Operators for Ontology Evolution. In: Meersman, R., Herrero, P., Dillon, T.S. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 544–553. Springer, Heidelberg (2009)
8. Magiridou, M., Sahtouris, S., Christophides, V., Koubarakis, M.: RUL: A Declarative Update Language for RDF. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 506–521. Springer, Heidelberg (2005)
9. Martin, M.: Exploring the netherlands on a semantic path. In: Auer, S., Bizer, C., Müller, C., Zhdanova, A. (eds.) Proceedings of the 1st Conference on Social Semantic Web, Leipzig, Germany, GI-edn., LNI, vol. P-113, p. 179. Bonner Köllen Verlag (2007) ISSN 1617-5468
10. Noy, N.F., Klein, M.C.A.: Ontology Evolution: Not the Same as Schema Evolution. Knowl. Inf. Syst. 6(4), 428–440 (2004)
11. Rosenfeld, M., Fernández, A., Díaz, A.: Semantic Wiki Refactoring. A strategy to assist Semantic Wiki evolution. In: Proceedings of the Fifth Workshop on Semantic Wikis (SemWiki 2010), co-located with 7th European Semantic Web Conference, ESWC 2010 (2010)
12. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: User-Driven Ontology Evolution Management. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS (LNAI), vol. 2473, p. 285. Springer, Heidelberg (2002)