JOPA: Developing Ontology-Based Information Systems

Martin Ledvinka, Petr Křemen

Czech Technical University in Prague, Czech Republic

Abstract. Accessing OWL ontologies programmatically by complex IT systems brings many problems stemming from ontology evolution, their open-world nature and expressiveness. This paper introduces Java OWL Persistence API (JOPA) that tackles these problems by providing Object-Ontological Mapping (OOM) driven by integrity constraints designed on top of OWL ontologies, together with transactional processing and various backend implementations. Based on the lessons learnt during the design and implementation of the prototype, we present a proposal for a two layered architecture of object-oriented ontological information system design and analyse complexity of various operations for object-oriented ontological applications.

1 Introduction

With the growing amount of knowledge available through (semantic) web, appropriate tools are necessary to author, process and search the knowledge. Ontology-editors are on one side – they are generic, ontology-independent, but often hardly understandable for domain experts not trained in knowledge modelling. Those people require information systems, business logic and user interface of which is tailored to the particular domain of their expertise.

Comparing to traditional information systems that are often built on top of relational databases, design of ontology-based information systems brings many challenges as well as advantages for information system designer [17]. On one side, information systems, that accept closed-world assumption by their nature, have to deal with distributed and open-world knowledge represented in ontologies, on the other hand, ontological changes often do not affect the information system data model assumptions and thus can be smoothly applied without information system recompilation and redeployment (e.g. taxonomy/metadata extension). Furthermore, expressive power of semantic web ontologies is significantly higher than that of relational databases.

In this paper, we review current work in the field of ontology-driven information system design, and discuss main ideas and implementation experience of the JOPA framework[1] for ontology-based information system access. Complementary to the works [18], [17] that dealt with the interaction between ontologies and information systems, we go deeper here and analyze requirements of an

[1] `http://sourceforge.net/projects/jopa`, cit. 16. 6. 2014

object-oriented ontology information system w.r.t. efficient ontological storage, with focus on OWL 2 ontologies [22].

Section 2 presents the relationship of our work to the state-of-art research. Section 3 introduces design and implementation of a prototype system for ontology-based information system access. Section 4 presents a proposal for the layered architecture for ontology-based information system access. The paper is concluded in Section 5.

2 Related Work

Significant research effort has been spent on programmatic access to RDF [10] and OWL ontologies. As already mentioned in [19], and [17], two types of information system architectures can be distinguished:

generic (domain-independent), where no assumptions are made about the ontology in compile-time. This approach typically uses some generic (type 1) API to ontologies like Jena [9], Sesame [8] (for RDF ontologies), or OWLAPI [15] (for OWL ontologies). Maintenance of such applications is very difficult. A change in the ontology (e.g. class/property addition/removal) might break the application logic in runtime. OWLlink [20], which is in essence a remote reasoner interface, belongs to this category as well.

domain-specific, where part of the ontological schema is compiled into the object model. Such approach significantly simplifies the work of a programmer, who can work with object-model counterparts of ontological entities directly. On the other hand ontological changes might break the object model and result even in compile-time errors. APIs (type 2) of this nature include Jastor [25], Alibaba [6] (for RDF), or Sapphire[24], JAOB[21], or Empire[14] (for OWL). Information system described in [27] is also a member of this group, being basically an ontology management system (similar to tools like PhpMyAdmin[2]) and using annotated Java classes merely as data transfer objects [12].

While the discussed programmatic access options involve many RDF as well as OWL libraries, OWL support is only available in a few ontological storages. One of the most advanced is Stardog [7], providing full sound and complete OWL 2 DL reasoning for ontological schemas (TBoxes). Another well-known OWL storage is OWLIM[23], being a plugin into the OpenRDF Sesame triple store[5]. OWLIM provides rule-based reasoning with *full materialization*, thus covering OWL2-RL as well as sound, but incomplete inference over other OWL subsets.

Many RDF stores, including OpenRDF Sesame [5], Virtuoso [4], SDB[1] or TDB[2], provide support for transactional processing.

[2] http://www.phpmyadmin.net, cit. 18. 6. 2014

3 JOPA

In this section, we first review basic notions and then introduce the main principles of JOPA design, together with its protypical implementation.

3.1 Background

We consider programmatic access to OWL 2 DL ontologies, corresponding in expressiveness to the description logic $\mathcal{SROIQ}(\mathcal{D})$[3]. In the next sections, consider an OWL 2 DL ontology $\mathcal{O} = (\mathcal{T}, \mathcal{A})$, consisting of a TBox $\mathcal{T} = \{\tau_i\}$ and an ABox $\mathcal{A} = \{\alpha_i\}$, where α_i is either of the form $C(i)$ (class assertion), or $P(i, j)$ (object property assertion), where $i, j \in N_i$ are OWL named individuals, $C \in N_c$ is a *named class*, $P \in N_r$ is a *named object property.* Other axiom types belong to $\mathcal{T}$. W.l.o.g. we do not consider $C(i)$ and $P(i, j)$ for complex C and P here. We do not consider anonymous individuals either. See full definition of OWL 2 DL [22] and $\mathcal{SROIQ}(\mathcal{D})$ [16].

In addition to ontological (open-world) knowledge, a *set* $\mathcal{S_C} = \{\gamma_i\}$ *of integrity constraints* is used to capture the contract between an ontology and an information system object model. Each integrity constraint γ_i has the form of an OWL axiom with closed-world (constraint) semantic, as defined in [26].

Example 1. Ontology and Integrity Constraints

$$\mathcal{O}_a = (\{Person \sqsubseteq Animal\}, \{hasFriend(Alice, Bob), hasFather(Alice, Cyril)\})$$
$$\mathcal{O}_b = (\{Person \sqsubseteq Animal\}, \{hasFriend(Alice, Bob), Animal(Bob), hasFather(Alice, Cyril)\})$$
$$\mathcal{S}_{\mathcal{C}1} = \{Person \sqsubseteq \forall hasFriend \cdot Person\}$$
$$\mathcal{S}_{\mathcal{C}2} = \{Person \sqsubseteq \forall hasFriend \cdot Animal\}$$

The ontology $\mathcal{O}_a$ states that each $Person$ is also an $Animal$ and that $Alice$ is a friend of Bob. First, while $\mathcal{O}_a$ is consistent, integrity constraints in $\mathcal{S}_{\mathcal{C}1}$ are violated as Bob is neither a known instance of $Person$ nor $Animal$. Furthermore, ontology $\mathcal{O}_b$ is also consistent, but in this case integrity constraint in $\mathcal{S}_{\mathcal{C}2}$ are satisfied (Bob is a known instance of $Animal$), while $\mathcal{S}_{\mathcal{C}1}$ are violated.

Let us also define additional ontology structures. By *multi-context ontology* we denote a tuple $\mathcal{M} = (\mathcal{O}_d, \mathcal{O}_1, ..., \mathcal{O}_n)$, where each $\mathcal{O}_i$ is an ontology identified by a unique IRI and is called *context*, $\mathcal{O}_d$ denotes the *default ontology (default context)* which is used when no other context is specified. This structure basically corresponds to an RDF dataset with named graphs [10]. An *ontology store* is the underlying backend for performing operations on $\mathcal{M}$.

[3] For the sake of compactness, in this paper we neglect datatypes and literals ($\mathcal{D}$) and use description logic notation.

3.2 Architecture

The two sets of integrity constraints from Example 1 can represent two different information systems, e.g. $\mathcal{S}_{\mathcal{C}1}$ for a social network, and $\mathcal{S}_{\mathcal{C}2}$ for some dog shelter web site. Each integrity constraint set represents the *contract* between the information system object model and the ontology, see [19] for more details. Once an integrity constraint is violated by the user-inputed data, the contract is broken and the information system object model must be adjusted. All other changes of the ontology are compliant with the information system and do not need information system adjustments. This allows easy validation of the compliance between the ontology and the information system logic by checking integrity constraints, even without actually passing the data to the information system.

From the data access point of view, JOPA approach lies between the Type 1 and Type 2 APIs (see Section 2), allowing to use precompiled object model for application-specific fixed schema of the ontology defined by the set of integrity constraints, while providing also access to the application-independent parts of the ontology. I.e. all properties of an individual (e.g. in Example 1, the $hasFather(Alice, Cyril)$ property assertion) can be obtained. As a result, full ontological reasoning results, including inferred axioms, are available to the information system designer.

Overal JOPA-based information system architecture is depicted in Figure 1.

3.3 Limitations of the Current Architecture

The current architecture of JOPA has several shortcomings. E.g., it is not able to exploit multi-context storages. It always works on the default context, without having any guarantee as to its contents.

The efficiency of access to the underlying ontology is also problematic. The storage access strategy highly depends on the underlying API and bulk operations support (e.g. load all entity attributes in one query) is currently very limited.

Another issue is transaction support on storage level. Preserving the same transactional behaviour across all supported storages is, given their variety, a difficult task. Take, for example, the difference between OWLAPI accessed OWL file, which has no transactional support, and an OWLIM repository, which does support storage-level transactions. JOPA tackles the problem by creating snapshots of the underlying ontology. This is obviously inefficient for large datasets and more advanced storages like Stardog or OWLIM offer better options (e.g. creating temporary contexts).

4 Proposed Architecture

The contribution of this paper is described in this section. Based on our experience with developing JOPA, we propose a two layered architecture. The layers essentially correspond to the roles of a JPA provider and a JDBC driver, but

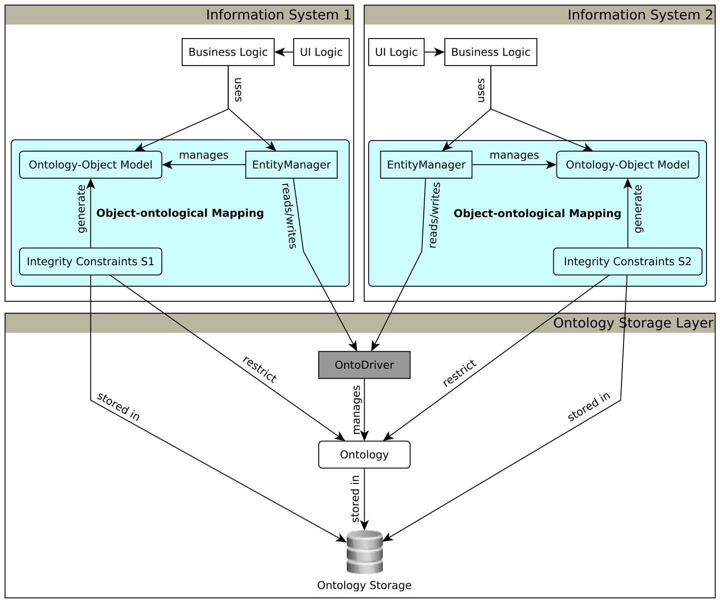

Fig. 1. JOPA Architecture. Two different information systems are depicted, each having its own set of integrity constraints and business and UI logic, but sharing the same ontological structure. Note that the OntoDriver is not part of the current architecture, it represents a part of our proposal.

their APIs have to reflect the characteristics of the underlying semantic data, for instance:

– Inferred attributes are effectively read-only, because they are not explicitly expressed in the underlying ontology and thus cannot be changed externally,
– As was stated in Section 3, user expresses the contract between the ontology and the object-model by means of integrity constraints. These constraints have to be verified accordingly,
– Besides explicitly stated attributes of entities, there are also properties that are not necessarily part of the object-model (see 3.2). However, JOPA has to be able to handle these property values the same way as property values expressed as entity attributes,
– Last, but not least, the API of JOPA has to be able to support multi-context storages, which have no equivalent in relational databases.

4.1 JOPA

The central point of interaction between the user application and JOPA is the `EntityManager` interface. This interface in JOPA mostly corresponds to its JPA counterpart, as defined by the JPA 2 specification [3]. However, to support multi-context storages, e. g. Sesame, we overload methods `find`, `persist` and `merge` with additional versions accepting an instance of class `EntityDescriptor`. This instance is used to provide provenance information about the entity and its attributes. Using `EntityDescriptor`, it is possible to specify contexts from which attribute values should be loaded.

If the provenance information isn't specified, JOPA uses the default context of the storage.

Storage Access Corresponding to the approach taken by JPA, each physical storage is represented by a single persistence unit. In the earlier version of JOPA all the storages were managed by the same persistence unit, forcing the user to specify which storage to use when executing methods on `EntityManager`. This mechanism polluted the API and made it difficult to understand and use.

The only trade-off is that one will not be able to specify values for entity attributes from different storages. However, we consider such situation less common and solvable e.g. using a federated storage.

4.2 OntoDriver

To unify access to various ontology storages we define the **OntoDriver**. This software layer will hide the differences between storage APIs behind a unified interface providing services optimized for object-oriented application access and will serve as an ontology data provider for OOM in JOPA. The most straightforward approach to designing such a layer would be to adopt the JDBC structure and realize the communication using OWL ontology queries and result sets. However, due to the lack of standardisation in the field of OWL 2 queries and since most contemporary storages provide an API for directly manipulating axioms as their primary interface, we have decided to exploit such interfaces. The main reasons for this are:

– Only a limited set of query types is required,
– Eliminating generation and parsing of SPARQL (or any other language) queries for data retrieval and manipulation,
– Exposing API suitable for typical operations performed by an OOM framework, i. e. reading and manipulation of property values of a single individual, which can be optimized.

Typical Operations To formally describe typical operations required of OntoDriver, we need two auxiliary structures: *entity descriptor* and *axiom descriptor*.

An entity descriptor δ_e is a tuple $(i, \{(r_1, b_1)...(r_k, b_k)\})$, where $i \in N_i$, $r_m \in N_r$, $b_m \in \{0, 1\}$ and $m \in 1...k$. The b_ms specify whether inferred values for the given role should be included as well.

An axiom descriptor δ_a is a tuple $(i, \{(r_1, v_1)...(r_k, v_k)\})$, where $i \in N_i$, $r_m \in N_r$, $v_m \in N_i$ and $m \in 1...k$. The v_m represent property assertion values for the given individual and property.

OntoDriver is required to support at least the following core operations:

– $find(\mathcal{M}, \delta_e)$: $2^{\mathcal{M}} \times N_i \times N_r^k \times \{0,1\}^k \rightarrow 2^{N_i \times N_r \times N_i}$, where δ_e is an entity descriptor,
  - Given an individual, load values for the specified properties,
– $persist(\mathcal{M}, \delta_a) = \mathcal{M} \cup \{\alpha_1 ... \alpha_s\}$, where $\alpha_1 ... \alpha_s$ are property assertion axioms created from role-value pairs in δ_a,
  - Persist axioms representing entity attribute values,
– $remove(\mathcal{M}, \delta_a) = \mathcal{M} \setminus \{\alpha'_1 ... \alpha'_t\}$, where $\alpha'_1 ... \alpha'_t$ are property assertion axioms for the roles specified in δ_a,
  - Remove axioms representing entity attribute values,
– $update(\mathcal{M}, \delta_a) = (\mathcal{M} \setminus \{\alpha'_1 ... \alpha'_t\}) \cup \{\alpha_1 ... \alpha_s\}$, where $\alpha'_1 ... \alpha'_t$ are original property assertion axioms for the roles $r_1 ... r_k$ defined in δ_a and $\alpha_1 ... \alpha_s$ are new property assertion axioms created for role-value pairs in δ_a,
  - Remove old and assert new values for entity attributes,
– $getTypes(\mathcal{M}, i, b)$: $2^{\mathcal{M}} \times N_i \times \{0,1\} \rightarrow 2^{N_c}$, where the resulting axioms represent types of the specified individual i, b specifies whether inferred types should be included as well,
  - Get types of the specified named individual,
– $addTypes(\mathcal{M}, i, \{c_1 ... c_k\}) = \mathcal{M} \cup \{\alpha_1 ... \alpha_k\}$, where $c_m \in N_c$ and the α_m are class assertion axioms for the given individual i,
  - Adds class assertion axioms for the given individual,
  - $removeTypes(\mathcal{M}, \{c_1 ... c_k\})$ is defined analogically,
– $validateIC(\mathcal{M}, \{\gamma_1 ... \gamma_k\}) : 2^{\mathcal{M}} \times 2^{N_i \times N_r \times N_i} \times \mathcal{S}_c \rightarrow \{0,1\}$, where $\gamma_m \in \mathcal{S}_c$ and $m \in 1...k$,
  - Validate the specified integrity constraints, verifying *reasoning-time* integrity constraints which cannot be validated at runtime [19].

Transaction Snapshots The issue of transaction snapshot size (see Section 3.3) can by tackled by extracting syntactic modules ([13]) based on the application object model. Syntactic modules are, for most ontologies, comparable in size with semantic modules, but their extraction complexity is quadratic, compared to undecidable complexity for semantic modules in more expressive languages, see [13] for more information about module extraction complexity.

More importantly, the OntoDriver layer enables us to introduce any different transactional strategy suitable for the underlying storage, as long as the expected behaviour is preserved.

The OntoDriver API The key element of the API for OntoDriver is the `Connection` interface, which provides methods for querying and manipulating data. The methods in the `Connection` interface can be divided into the following groups:

– Standard housekeeping methods like `commit`, `rollback`, `close`, `isOpen` etc.,
– Methods for SPARQL[4] queries, e. g. `createStatement`,

[4] In OWL 2 DL entailment regime. However, the API itself does not restrict the query language.

– Methods for working with contexts, e. g. `getContexts` or `isConsistent`
– Methods corresponding to operations defined in Section 4.2

The interface defines, on one hand, operations that are highly optimized in existing reasoners (e. g. consistency check, individual's types retrieval). Most of the methods represent, however, operations designed specifically for application access, where it is necessary to manipulate knowledge related to an individual efficiently. For instance, when compared to OWLlink, where retrieval of entity attributes would require for each attribute separate `GetDataPropertyTargets` or `GetObjectPropertyTargets` [20] requests, in OntoDriver we are able to perform such operation in one method call.

4.3 Optimized Ontological Storage

Given the rather specific needs of OOM frameworks, represented by the API of OntoDriver, we can even propose a data storage optimized for such needs. Contemporary ontology storages are mostly optimized for fast data retrieval, falling into the category of OLAP[5] knowledge storages. The most advanced current triple stores also contain, to facilitate fast query answering, several indexes, which increase data duplication and slow data manipulation down.

Optimizing Storage for Application Access Since we are dealing primarily with OWL ontologies, our ultimate goal would be a storage with OWL 2 DL reasoning support. As an inspiration for the possible optimized repository we have taken Parliament[6], which is a RDF triple store with full materialization and rule-based reasoning. The reasons why we chose Parliament are:

– The index structure as is is suitable for application access,
– The index used in Parliament produces almost no data duplication,
– Its indexing structure is fully disclosed and was elaborated in [11].

To give a short overview of Parliament's index, it is based on linked lists of statements with the same subject/predicate/object. Each statement contains a reference to the next statement with the same subject/predicate/object. This structure is very favourable for programmatic access, since loading an entity in essence requires just traversing list for the given subject and setting values based on the predicate.

Using doubly linked list The linked list structure can be suboptimal in cases where the size of the list is large. To improve performance of value lookup in such list, we propose using a doubly linked list, which would enable concurrent search of the list from both its ends.

Table 4.3 compares operation complexity in the general case without any index, in case of Parliament and for the optimized storage using doubly-linked lists. Please note that we are comparing the average case complexity (see [11] for difference between average and worst case complexity).

[5] Online analytical processing.
[6] http://parliament.semwebcentral.org/, cit. 18. 6. 2014

Operation	General	Parliament	Parliament-optimized
$find(\mathcal{M}, \delta_e)$	$O(\lvert\mathcal{M}\rvert)$	$O(min(len_s, len_p))$	$O(\frac{min(len_s,len_p)}{2})$
$update(\mathcal{M}, \delta_a)$	$O(\lvert\mathcal{M}\rvert \times 2)$	$O(min(len_s, len_p, len_o) \times 2)$	$O(min(len_s, len_p, len_o))$
$persist(\mathcal{M}, \delta_a)$	$O(\lvert\mathcal{M}\rvert)$	$O(min(len_s, len_p, len_o))$	$O(\frac{min(len_s,len_p,len_o)}{2})$
$remove(\mathcal{M}, \delta_a)$	$O(\lvert\mathcal{M}\rvert)$	-	$O(\frac{min(len_s,len_p,len_o)}{2})$

Table 1. Operation asymptotic complexity comparison. len_s, len_p and len_o denote the lengths of lists of statements with the same subject, predicate and object respectively. It can be seen that the complexity of all operations generally depends on the complexity of an axiom find. This is mostly due to the fact that insertion into the respective linked lists has constant complexity. Note that operations $getTypes(\mathcal{M}, i, b)$, $addTypes(\mathcal{M}, i, \{c_1...c_k\})$, $removeTypes(\mathcal{M}, \{c_1...c_k\})$ can be realized using the above ones.

5 Conclusions

We presented several ideas about the design of ontology-based information systems that were based on the experience gained during prototyping of the original JOPA system. We found out that access to ontological storages needs to be optimized w.r.t. the needs of object-oriented information systems. Basically, optimizing CRUD[7] operations of "frames" (objects) on the storage is crucial for efficient object-oriented access to OWL ontologies. For this purpose we defined an API (similar to JDBC) that encapsulates these optimized operations together with common transaction support, and analyzed the complexity of these operations.

In the next work, we will focus on empirical evaluation as well as transaction support optimization.

Acknowledgements This work was supported by the grant No. SGS13/204/OHK3/3T/13 Effective solving of engineering problems using semantic technologies of the Czech Technical University in Prague.

References

1. Apache Jena - SDB - persistent triple stores using relational databases. online, `https://jena.apache.org/documentation/sdb/`
2. Apache Jena - TDB. online, `http://jena.apache.org/documentation/tdb/`
3. JSR 317: JavaTM Persistence API, Version 2.0
4. OpenLink Virtuoso Univeral Server. online, `http://virtuoso.openlinksw.com/`
5. OpenRDF Sesame. online, `http://www.openrdf.org/`
6. openRDF.org: AliBaba. online, `http://www.openrdf.org/alibaba.jsp`
7. Stardog: The RDF Database. online, `http://stardog.com/`
8. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proceedings of the First International Semantic Web Conference on The Semantic Web. pp. 54–68 (2002)

[7] Create, Retrieve, Update, Delete.

9. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: Proceedings of the 13th international World Wide Web conference (Alternate Track Papers & Posters). pp. 74–83 (2004)
10. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. Tech. rep., W3C (2014)
11. Dave Kolas, I.E.M.D.: Efficient Linked-List RDF Indexing in Parliament. In: The Semantic Web - ISWC 2009 (2009)
12. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (2002)
13. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Just the Right Amount: Extracting Modules from Ontologies. In: Proceedings of the 16th international conference on World Wide Web (2007)
14. Grove, M.: Empire: RDF & SPARQL Meet JPA. online (April 2010), `http://semanticweb.com/empire-rdf-sparql-meet-jpa_b15617`
15. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web – Interoperability, Usability, Applicability (2011)
16. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. In: Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). pp. 57–67
17. Křemen, P.: Building Ontology-Based Information Systems. Ph.D. thesis, Czech Technical University, Prague (2012)
18. Křemen, P., Kostov, B.: Expressive OWL Queries: Design, Evaluation, Visualization. International Journal on Semantic Web and Information Systems 8(4), 57–79 (December 2012), `http://www.igi-global.com/viewtitlesample.aspx?id=75774&ptid=59645&t=expressive+owl+queries%3a+design%2c+evaluation%2c+visualization`
19. Křemen, P., Kouba, Z.: Ontology-Driven Information System Design. IEEE Transactions on Systems, Man, and Cybernetics: Part C 42(3), 334–344 (May 2012), `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6011704`
20. Liebig, T., Luther, M., Noppens, O., Wessel, M.: OWLlink. Semantic Web – Interoperability, Usability, Applicability 2(1), 23–32 (2011)
21. von Malottki, J.: JAOB (Java Architecture for OWL Binding). online, `http://wiki.yoshtec.com/jaob`
22. Motik, B., Parsia, B., Patel-Schneider, P.F.: OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3C recommendation, W3C (Oct 2009), http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/
23. Ontotext: OWLIM — Ontotext. online, `http://www.ontotext.com/owlim`
24. Stevenson, G., Dobson, S.: Sapphire: Generating Java Runtime Artefacts from OWL Ontologies. In: CAiSE Workshops. pp. 425–436 (2011)
25. Szekely, B.H., Betz, J.: Jastor Homepage, `http://jastor.sourceforge.net`
26. Tao, J., Sirin, E., Bao, J., McGuinness, D.L.: Integrity Constraints in OWL. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010), `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1931`
27. Zviedris, M., Romane, A., Barzdins, G., Cerans, K.: Ontology-Based Information System. In: Semantic Technology (2013)