Introducing Layers of Abstraction to Semantic Web Programming [1]

Bernhard G. Humm, Alexey Korobov

Hochschule Darmstadt – University of Applied Sciences,
Haardtring 100, 64295 Darmstadt, Germany
Bernhard.Humm@h-da.de

Abstract: Developers of ontologies and Semantic Web applications have to decide on languages and environments for developing the ontology schema, asserting statements, specifying and executing queries, specifying rules, and inferencing. Such languages and environments are not well-integrated and lack common abstraction mechanisms. This paper presents a concept framework to alleviate those problems. This is demonstrated by a complex sample application: reasoning over business process models.

Keywords: Semantic Web, software engineering, business process models, Lisp, Prolog

1 Introduction

Developers of ontologies and Semantic Web applications often face the following questions:

- Which language and environment shall be used for *developing the ontology schema*? For example, when developing a Semantic Web application for reasoning over business process models, the schema may include classes like "BusinessProcess" and "Activity" and properties like "hasBusinessProcess". In many projects, a graphical ontology development environment like Protégé or TopBraid Composer is chosen to develop the ontology schema in languages such as RDF, RDFS, and OWL.
- Which language and environment shall be used for *asserting statements*? Example in the business process domain: "CompanyABC hasBusinessProcess TravelManagement". While such statements may be added manually to an ontology in a graphical ontology environment, they usually have to be asserted dynamically by an application. For this, Semantic Web frameworks are used which allow to embed Semantic Web languages such as RDF, RDFS, and OWL into general purpose programming languages like Java. Examples for mainstream Semantic Web frameworks are Jena and Sesame.

[1] This research was funded by Zentrum für Forschung und Entwicklung (ZFE), Hochschule Darmstadt – University of Applied Sciences under grant number 419 327 01.

- Which language and environment shall be used for *specifying and executing queries*? Example: "Which companies have business processes for travel management?" Semantic Web query languages like SPARQL allow for querying ontologies. Where SPARQL queries usually can be executed from graphical ontology development environments, this usually is used for demonstration and testing purposes only. As with asserting statements, queries usually have to be executed dynamically from a Semantic Web application and the query results are processed further, e.g. displayed on a Web page. Again, Semantic Web frameworks allow to embed Semantic Web query languages such a SPARQL and further process the results.
- Which language and environment shall be used for *specifying rules and inferencing*? Example rule: "If two business processes have similar names then they are likely to be in the same business domain". Standardized Semantic Web rule languages like RIF or proprietary rules languages like Sesame Rules may be embedded in Semantic Web frameworks.

These questions lead to a number of issues or problems.

- *Developing ontology schema*: Manually developing an ontology schema in RDF/XML is not feasible due to its most verbose syntax. N3 is the most concise and suitable syntax for developing ontology schemas manually. However, it still lacks abstraction mechanisms. Every statement must be developed in the simple triple notation. The only grouping mechanisms are semicolon and comma notations for avoiding repetitions of subjects and subject / predicate combinations. For example, it is not possible to define a concept "reification" with three input parameters (subject, predicate, object) and a blank node as an output parameter. Instead, each reification requires the (redundant) specification of three triples with predicates `rdf:subject`, `rdf:predicate`, and `rdf:object`.
  Graphical ontology development environments largely alleviate those problems, for instance, by providing wizards for reifying statements. However, developers cannot define similar abstractions on their own.
- *Asserting statements*: a single triple to be inserted into a RDF store via deserializing RDF / RDFS / OWL takes a single line of N3 code. In contrast, adding the same triple programmatically via a Semantic Web framework like Jena or Sesame takes about 15 lines of Java Code[2]. This includes repository and connection handling, instantiating objects for resources and literals, asserting statements, and exception handling. So, simple statements cannot be expressed in a concise way as is possible in N3.
  On the other hand, Java offers mechanisms for defining abstractions such as classes and methods. A method "reify" taking "subject", "predicate", and "object" as input parameters and returning a reified blank node may be implemented once and then used many times wherever reification is needed in this form.
- *Specifying and executing queries*: SPARQL is a concise query language, similar to N3. However, it, too, misses abstraction mechanisms. Where SPARQL V1.1 intro-

[2] Basis: code samples from the Sesame User Guide
http://www.openrdf.org/doc/sesame2/users/ch08.html

duces subqueries, it still lacks named, parameterized queries that can be invoked as subqueries. For example, a query for all companies that provide business process ?x cannot be defined once and then re-used in many queries. Copying and pasting similar SPARQL WHERE parts is, therefore, common practice, leading to redundant code which is difficult to maintain.

Also, executing SPARQL queries from within Semantic Web frameworks is cumbersome. For example, in Sesame it takes about 20 lines of Java code to evaluate a one-line SPARQL query. This includes connection handling, query preparation and evaluation, iterating result set, identifying individual results, and exception handling.

- *Specifying rules and inferencing:* While SPARQL is *the* standard Semantic Web query language, a de-facto Semantic Web rule standard has not yet emerged. Embedding rules in Semantic Web frameworks faces the same usability issues as asserting statements and executing queries.

In total, different languages and environments, all with their strengths and weaknesses, but not well integrated in a useable fashion impede developing ontologies and Semantic Web applications. In particular, abstraction mechanisms in Semantic Web languages and technologies are limited. We have developed a framework for concepts that addresses those issues – see the following section.

2 A Concept Framework for Semantic Web Programming

2.1 Environment

For the implementation of the concept framework, we have chosen AllegroGraph, a commercial Semantic Web framework by Franz Inc.. AllegroGraph is based on Lisp, in particular Allegro Common Lisp, a professional implementation of the ANSI Common Lisp standard. It supports RDF, RDFS, SPARQL, and the OWL subset RDFS-Plus.

AllegroProlog is used as reasoning and query language. AllegroProlog is a Prolog implementation by Franz Inc., fully integrated in Lisp. It allows Prolog programming in Lisp notation.

It shall be noted, however, that the concept framework described in this paper is not specific to Lisp, Prolog, AllegroGraph, or AllegroProlog.

2.2 Our Use of the Term "Concept"

Encyclopedia Britannica defines *concept* as: "an abstract or generic idea generalized from particular instances" [1]. This definition is valid for our purposes. Additionally, our notion of concept is always in the context of an application domain for which an ontology or a Semantic Web application is developed. For example, in the application domain of business process models, *UML activity* [2] is a concept.

2.3 A DSL for Specifying Concepts

We have developed a simple Domain-Specific Language (DSL) [3], called *concept DSL*, that allows for specifying concepts. The *basic features* of the concept DSL are as follows:

- `define-concept` specifies a new named concept with *0..n* concept parameters.
- `triple` allows for using all provided RDF, RDFS, and OWL constructs as well as self-defined classes, instances, and properties.
- `<concept>` allows for using all lower-level, more concrete concepts previously defined via `define-concept` by their names.

In summary, concepts form trees with triples as leaves and other concepts as inner nodes.

Advanced features of the concept DSL are as follows:

- `&optional` allows specifying optional concept parameters.
- `local` allows using local variables within concept specifications. Local variables are particularly useful for introducing blank nodes and for generating URIs.
- `cond` allows for specifying pre-conditions to be checked before asserting statements, reasoning, and querying.
- `<name space>:<concept name>` allows for using identical concept names in different name spaces for different application contexts. They support the development of large ontologies. Where used, name space identifiers precede a concept name, separated by a colon.

In summary, a concept specification in a BNF-like notation [4] is as follows:

```
(define-concept <concept> (<parameter>*
                           [&optional <parameter>*])
  [(local <variable> <expression>)]
  [(cond <expression>)]
  ([<namespace>:]<concept> <parameter>*)*
  (triple <subject> <predicate> <object>)*)
```

Example: concept of an RDFS instance

```
(define-concept instance (uri class &optional label comment)
  (triple uri !rdf:type class)
  (triple uri !rdfs:label label)
  (triple uri !rdfs:comment comment))
```

The concept specification uses `triple` only. The exclamation mark (Wilbur reader macro) indicates a Semantic Web URI.

Example: concept of a node in a graph:

```
(define-concept node (uri node-type graph label
                                          &optional comment)
  (cond (sub-class node-type !modl:node))
  (instance uri node-type label comment)
  (triple graph !modl:contains uri))
```

The higher-level, more specific concept `node` uses the lower-level, more general concepts `instance` and `sub-class`.

2.4 Framework Implementation

At compile time, the concept framework parses each concept specification and generates the following source code.

1. *Lisp function for asserting statements*: The function has the same name and parameters as the concept. Optional concept parameters are being handled using Common Lisp's `&optional` feature. Local concept variables are being handled via Lisp variables. The expression following `cond` are implemented as pre-conditions. Triples with respective subject, predicate, and object are asserted to the triple store using the AllegroGraph built-in function `add-triple`. For lower-level concepts being used, the respective Lisp function is invoked and the parameters are passed.
2. *Prolog predicates for reasoning and querying*: The predicates are named after the concept and contain all mandatory parameters. Since Prolog does not support optional parameters, optional concept parameters are being handled by generating multiple predicates with increasing numbers of parameters. Local concept variables are handled via Prolog variables. Expressions following `cond` are implemented as conjunctive goal terms. Different predicates allow for reasoning and querying with and without local variables. The built-in AllegroProlog predicate `q-` is used to prove against asserted triples in the triple store. For lower-level concepts being used, the respective Prolog predicate with its parameters is used as a conjunctive goal term.

Fig. 1 illustrates concept specification and generated Lisp function and Prolog predicates by the example of the concept node.

In the example, the Lisp function `node` is generated from the concept `node` and may be used as follows.

```
(node !trv:req-app !uml:activity !trv:uml "request approval")
```

The example shows a statement about a node in an UML activity diagram.

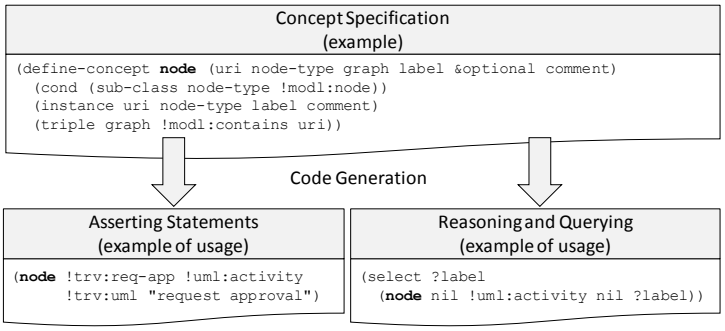

Fig.1: Concept definition and code generation

A query using the generated Prolog predicate `node` may look like this:

```
(select ?label (node nil !uml:activity nil ?label))
```

The query returns the labels of all nodes of type `!uml:activity`, in this case "`request approval`". `nil` indicates a don't care parameter value, e.g., the node's graph is irrelevant in this query.

The implementation of the concept framework is straight forward and comprises only about 100 lines of Lisp code excluding comments and blank lines. The core is the Lisp macro `define-concept` which generates Lisp code at compile time using the built-in Common Lisp macro processor.

3 Application: Reasoning over Business Process Models

We have applied the concept framework in a complex Semantic Web application that allows reasoning over business process models. In this section, we explain a sample application scenario, give an overview of the application, and show simple examples.

3.1 Application Scenario

Consider the following application scenario: Two companies decided to merge. To leverage synergies, their business processes shall be aligned. Business processes are modeled in numerous models in different formats in both companies, e.g., as UML activity diagrams, Event-Based Process Chain (EPC) diagrams, and Business Process Modeling Notation (BPMN) diagrams. The task of the application is to pre-select similar business process models to support human experts in their detailed analysis.

For this, we transform different business process models into an ontology and use reasoning mechanisms for detecting similarity between models.

3.2 Sample Business Processes

Consider, e.g., the process models for business travels in Fig. 2, one represented as an EPC diagram and the other one as a UML activity diagram. Both diagrams represent business processes for business travels – similar in content, but different in the modeling notations used as well as in details.

3.3 Language Stack and Layers of Abstraction

DSL stacking [7] is a form of layering where higher-level, more specific DSLs are implemented on lower-level, more general DSLs. The concept framework is designed to enable DSL stacking in Semantic Web applications. See Fig. 3 for the language stack of the business process reasoning application.

Allegro Common Lisp is the base language in which AllegroGraph and AllegroProlog are implemented. The concept framework uses functionality of both libraries. Using the concept framework, a layered set of concepts is defined for the application domain, business process models: concrete modeling notations like UML activity diagrams and EPC diagrams on top of general graph based models on top of general Semantic Web concepts. Concrete reasoning applications can be implemented using those concepts.

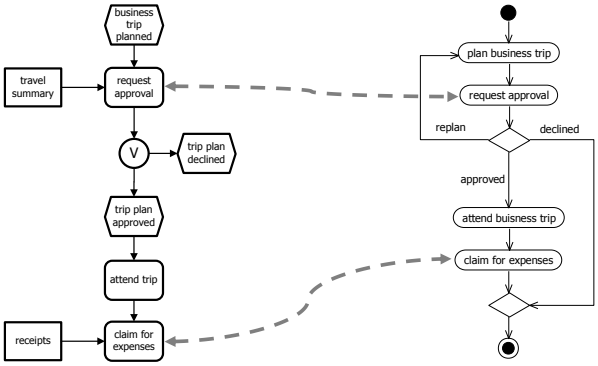

Fig. 2: Example business processes as EPC and UML activity diagram

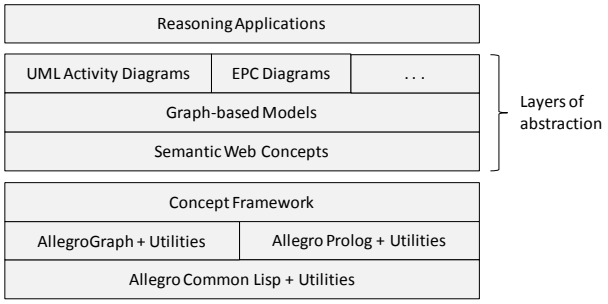

Fig. 3: Language stack

3.4 Concepts

Fig. 4 gives an overview of concepts being defined in the various layers of the business process reasoning application. One example concept, UML activity, is zoomed out.

UML activity is implemented using the general graph concept of a node. Node itself is implemented using the Semantic Web concept of an instance. Instance is implemented using triple from the concept framework.

The concept `activity` is defined as follows:

```
(define-concept activity(uri label diagram &optional comment)
  (node uri !uml:activity diagram label comment))
```

With the concept `activity` defined on top of the concept `node`, the creation of a UML activity node can be expressed more concisely as in Section 2.4 as follows:

```
(activity !trv:req-app "request approval !trv:uml)
```

Using the concept `follows`, the edges of the UML activity diagrams can be asserted, e.g.,

```
(follows !trv:split !trv:req-app)
```

This code for asserting statements is typically generated, for example, from the XML output of a UML tool.

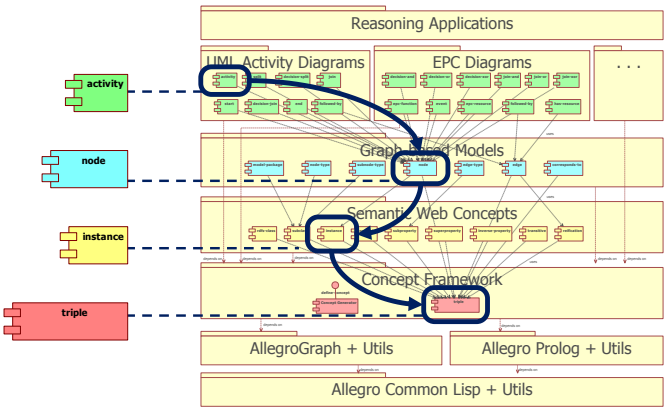

Fig. 4: Sample concepts

3.5 Querying and Reasoning

Asserted statements can conveniently be queried using the Prolog predicate generated from the concept specification – see the example in Section 2.4. In this section, we show how the concept framework supports the development of reasoning applications.

Determining the similarity between business process models is a complex task. We can rate similarity between two business process models concerning different aspects.

- *Diagram*: similarity of diagram titles and diagram types
- *Nodes*: similarity of node names and node types
- *Structure*: similarity of edge structures, e.g., similar nodes following other similar nodes

The following sample AllegroProlog rule detects a simple aspect of similarity between nodes, namely identical labels.

```
(<- (label-equivalence ?label ?n1 ?n2 ?d1 ?d2)
  (node ?n1 nil ?d1 ?label)
  (node ?n2 nil ?d2 ?label))
```

The rule consists of the *head term* `(label-equivalence …)` and the *goal terms* `(node …)`. If all goal terms can be proven then the head term is proven. The example reads like this: if two nodes `?n1` and `?n2` can be found in two diagrams `?d1` and `?d2` and their labels are identical (`?label`) then `label-equivalence` is true. For this predicate, the predicate `node` is used. It is satisfied no matter whether the nodes are UML activities, EPC functions or of any other node type. This is expressed by using `nil` as `node-type` parameter. The identity of the labels is assured via unification of the variable `?label`.

The following query selects all labels that occur as identical node names in any two diagrams.

```
(select ?label (label-equivalence ?label ?n1 ?n2 ?d1 ?d2))
```

Assuming that all nodes of the EPC diagram and the UML activity diagram in Fig. 2 have been asserted, the result is

```
("request approval" "claim for expenses")
```

When providing different parameters to the select statement, the following queries may be formulated using the predicate `label-equivalence`, all consisting of a single line of code in the query body:

1. Find equivalent labels in two specified diagrams
2. Instead of the labels, select the URIs of the nodes with identical labels in two diagrams
3. Select all diagrams in which there are identical node labels as in a given diagram
4. Select all pairs of diagrams with equivalent labels
5. Select all duplicates within one diagram
6. Select all duplicates within all diagrams

Many more types of queries may be expressed with this simple Prolog predicate.

We now present a more complex sample predicate to reason over structural similarity between diagrams.

```
(<- (succ-equivalence ?src1 ?src2 ?dest1 ?dest2 ?d1 ?d2)
  (label-equivalence ?lsrc ?src1 ?src2 ?d1 ?d2)
  (label-equivalence ?ldest ?dest1 ?dest2 ?d1 ?d2)
  (follows-transitively ?dest1 ?src1)
  (follows-transitively ?dest2 ?src2))
```

The predicate checks for successor equivalence, i.e., whether two pair wise equivalent nodes in two models follow each other – either directly or indirectly (`follows-transitively`). The possibilities of different meaningful queries using predicate `succ-equivalence` are even greater than the ones for `label-equivalence`. There is an enormous amount of different meaningful queries, each with a single line of code in the query body using the predicate `succ-equivalence`.

4 Evaluation

4.1 Comparison with Semantic Web Technologies

We now compare implementing ontologies and Semantic Web applications with and without using the concept framework.

For asserting statements consider, again, the example UML activity (Section 4.4). In N3, the equivalent assertion would be as follows:

```
trv:req-app rdf:type uml:activity;
            rdfs:label "request approval".
trv:uml modl:contains trv:req-app.
```

The assertion using the concept `activity` is more concise: 1 loc (line of code) compared to 3 loc in N3 notation, respectively 9 loc in N-triples notation, 15 loc in RDF/XML, and about 15-20 loc in the Semantic Web framework Sesame.

The concept `activity` hides implementation details: the use of class `uml:activity` and the use of properties `rdf:type`, `rdfs:label`, and `modl:contains`.

For comparing queries, consider the simple example of querying for label equivalence in Section 3.5. In SPARQL, the equivalent query would be as follows:

```
SELECT ?label WHERE {
  trv:uml modl:contains ?node1.
  trv:epc modl:contains ?node2.
  ?node-type1 rdfs:subClassOf modl:node.
  ?node1 rdf:type ?node-type1;
         rdfs:label ?label.
  ?node-type2 rdfs:subClassOf modl:node.
  ?node2 rdf:type ?node-type2;
         rdfs:label ?label.}
```

The WHERE part of the query comprises 8 lines of SPARQL code. In comparison, the WHERE part of the Prolog query using `label-equivalence` comprises 1 loc – with the definition of `label-equivalence` comprising 3 loc. Many other queries like, for example, selecting duplicates within diagrams (for more examples, see Section 3.5) may be formulated using `label-equivalence` – the WHERE part always being a one-liner. Using SPARQL, every single WHERE part has to be programmed individually and would comprise about 8 loc each time. Invoking the SPARQL query from a Semantic Web framework like Sesame adds about another 20 loc each.

Comparing more complex assertions and queries, the differences in code size even get larger. For example, one SPARQL query for the successor equivalence example from Section 3.5 comprises 16 loc compared to the one-liner in the Prolog query using `succ-equivalence` (definition: 5 loc). Varying the parameters passed, dozens of different Prolog queries may be formulated using `succ-equivalence` – with 1 loc each. Every equivalent SPARQL query would comprise about 16 loc (SPARQL) plus about 20 loc (Java) in the Semantic Web framework each.

4.2 Related Work

Many publications discuss the layering of Semantic Web languages (e.g., [5], [6]). Whereas those publications focus on the expressive power of the underlying languages and mechanisms, we take a Software Engineering view focusing on the ontologies and application code to be developed.

Humm and Engelschall introduce a method called DSL stacking [7]. According to the paradigm of Language-Oriented Programming, an application for a problem should

be implemented in the most appropriate domain-specific language (DSL). DSL stacking is a method for implementing Language-Oriented Programming where DSLs are incrementally developed on top of each other thus providing layers of abstraction. The concept framework can be seen as a DSL stacking platform for the Semantic Web.

Knublauch introduces the SPARQL Inferencing Notation (SPIN, [8]). He also identifies the lack of abstraction mechanisms in Semantic Web technology. SPIN targets the same goals as the concept framework. It introduces so-called SPIN functions that are SPARQL functions which can be used in FILTER or LET statements. SPIN Templates are re-usable SPARQL queries that can be instantiated with parameters. SPIN templates can be used instead of typing in SPARQL queries by hand. Thus, SPIN functions and templates introduce named queries that can be invoked as subqueries which is not possible in the current SPARQL standard. The crucial difference to the concept framework is that asserting statements and querying are handled separately. In the concept framework, one concept definition may be used for asserting statements and reasoning / querying.

In [9], Eiter et al. present an approach for integrating rules and ontologies in the Semantic Web. The approach combines answer set programming with description logics. The rules being used are similar to Prolog rules with negation as failure. They may contain queries. Eiter et al. claim an encapsulation view that increases flexibility to be one advantage of their approach. Our approach of using Prolog rules in Semantic Web applications has a similar aim. It also allows for reasoning and querying. It fosters encapsulation and we also claim increased flexibility as a result. While Eiter et al. provide a sound conceptual basis for integrating rules and ontologies, however, they do not cover abstraction and encapsulation mechanisms for asserting statements.

In [10], Le-Pouc et al. identify limited software support and the lack of standard programming paradigms in Semantic Web standards. To alleviate those problems, they introduce Semantic Web Pipes to support fast implementation of semantic data mash-ups while preserving abstraction, encapsulation, component-orientation, code re-usability, and maintainability. They present piping operators including the CONSTRUCT and SELECT operators that allow using results of SPARQL queries to be used in further processing. We do agree with their analysis of the state-of-the-art in Semantic Web programming. Their solution has similarities with our solution in that higher-level, more specific named constructs (here: pipes) can use lower-level, more general constructs. Again, as with [8] and [9], the difference to our approach is the sole focus on querying, not on asserting – thus solely focusing on the Semantic Web application developer and not, additionally, on the ontology modeler.

5 Conclusions

We have presented a novel approach for realizing layers of abstraction in ontology modeling and Semantic Web applications. A concept framework allows specifying higher-level, more specific concepts on top of lower-level, more general concepts. From concept specifications, code for asserting statements (Lisp functions) as well as for reasoning and querying (Prolog predicates) is being generated. We have shown that using the concept framework considerably reduces code size of ontologies and Semantic Web applications. The amount of code savings can fairly be considered an

order of magnitude. Using the concept framework furthermore enhances software quality regarding conciseness, understandability, and maintainability of the resulting code.

We have successfully used the concept framework in a complex application domain: reasoning over business process models. In this article, we have only shown a small and simple exemplary subset of the ontology and sample predicates for reasoning similarity. Additionally, we have implemented similarity metrics including linguistic analyses such as synonym resolution. In addition to similarity, we provide predicates for checking consistency of business process models and conformance to reference architectures. Our application not only covers the application scenario of mergers and acquisitions, but also the broader application scenario of e-business integration.

So far, our development is in the research and prototyping stage. Current and future work includes the following:

- Improving the stability of the concept framework
- Providing means for dedicated performance optimization of generated predicates where necessary
- Porting the concept framework to Semantic Web environments based on mainstream platforms like Java

We feel that our approach could, eventually, considerably improve the way ontologies and Semantic Web applications are developed in the future.

6 References

[1] Encyclopedia Britannica Ultimate Reference Suite 2010

[2] Havey, M.: Essential Business Process Modeling. O'Reilly Media, Sebastopol, CA (2005)

[3] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices, 35, 26—36 (2000)

[4] Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In: Proceedings of the International Conference on Information Processing, pp. 125—132. UNESCO (1959)

[5] Horrocks, I., Parsia, B., Patel-Schneider, P., Hendler, J.: Semantic Web Architecture: Stack or Two Towers?, Springer Berlin / Heidelberg (2005), pp. 37-41

[6] Kifer, M., de Bruijn, J., Boley, H., Fensel, D.: A Realistic Architecture for the Semantic Web, Springer Berlin / Heidelberg (2005), pp. 17-29

[7] Humm, B., Engelschall, R.: Language-Oriented Programming via DSL Stacking. In: Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT 2010), pp. 279—287. Athens, Greece (2010)

[8] Knublauch, H.: The Object-Oriented Semantic Web with SPIN. http://composing-the-semantic-web.blogspot.com/2009/01/object-oriented-semantic-web-with-spin.html (2009)

[9] Eiter, T, Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. In Proc. KR2004. AAAI Press (2004)

[10] Phouc, D. L., Polleres, A., Morbidoni, C., Hauswirth, M., Tummarello, G.: Rapid Prototyping of Semantic Mash-Ups through Semantic Web Pipes. In Proceeedings of the 18th International World Wide Web Conference (WWW2009), ACM, Madrid, Spain (2009)