Ontology structure evolution in a framework for building ontology based information systems

Samuil Nikolov

Abstract: *The paper describes the way ontology evolution is implemented in a framework for building ontology based information systems. The evolution is done in two phases – trivial class transformation and rule-based transformation. The chosen structure of the framework and the way ontology data is stored in its data base facilitate the easy implementation of the described approach. The structure evolution supports automatic and on-demand updates of old class instances stored in the database when a new class version is available.*

Key words: *Ontologies, Information Systems, Schema Update*

INTRODUCTION

Ontology structure evolution can be related to the problems of database and XML schema evolution. The actual problem consists in applying the new structural changes to the already stored data or data, created using the old schema. It is most often solved by isolating the applications from changes in every possible way – most often by using a stable database schema or by using views on the database tables [4]. This strategy requires predicting all eventual future data field needs which is impossible in most cases. Some of the notable existing solutions are summarised as follows:

- Ra and Rundensteiner [11] solve the problem by using view schema evolution approach. It computes a new view schema that reflects the semantics of the desired schema change, and replaces the old system's view with the new one;
- Xuan et al. [13] assume that changes of an ontology will not make false axioms that were previously true. Using this assumption, they offer a solution to ontology asynchronous versioning problem by storing old versions' data in separate tables that are used only by particular version or a set of versions of the application they present;
- Noy et al. [8] as well as Stojanovic et al. [12] present solutions to the ontology versioning problem where the two ontologies are compared, the changes are categorised and presented to the end user for confirmation. This approach is suitable for the ontology editors reviewed in their publications - Protege [15] and KAON [16] respectively, but not suitable when ontologies are used for other purposes as is the case with the current publication;
- Papastefanatos et al. [9] present an extension to structured query language (SQL) that allows additional declarations when creating the database structure. These are used to specify how eventual future changes to those elements should be treated.

The goal of this publication is to share experience in automatic updating of a system's data. The proposed update procedure allows continuous user operation in the system. The publication actually describes the way software configuration management [2] is implemented in the framework for building ontology-based dynamic applications described in [7] - the way new versions of generated information systems are deployed. The framework uses a set of models – domain specific language (DSL) programs - each representing a single class of an ontology. They are handled by a core system implemented in C++ and an integrated production system interpreter – in our case CLIPS production system.

The DSL programs contain declarations of class properties, information about the way they should be visualized and a set of rules that validate them and implement the information system's business logic. The framework's database is organized as a multi-tier vertical table [1] associating every class instance data to a unique identifier. One of the vertical tables contains simple ontology class properties – the ones with cardinality equal to one. Another two tables contain metadata and data of the complex properties – the ones with larger cardinality. Those are presented to the end user of the framework as grid controls. The framework's database contains the assertional component (ABox) of the ontology data, while the terminological component (TBox) is represented by the mentioned DSL programs, one for every ontology class.

1. Transformation of ontology class instances

1.1. Overview of the solution

Figure 1 shows an overview of how the framework handles ontology class change. It shows what happens when a class X_1 is replased by its next version – X_1'.

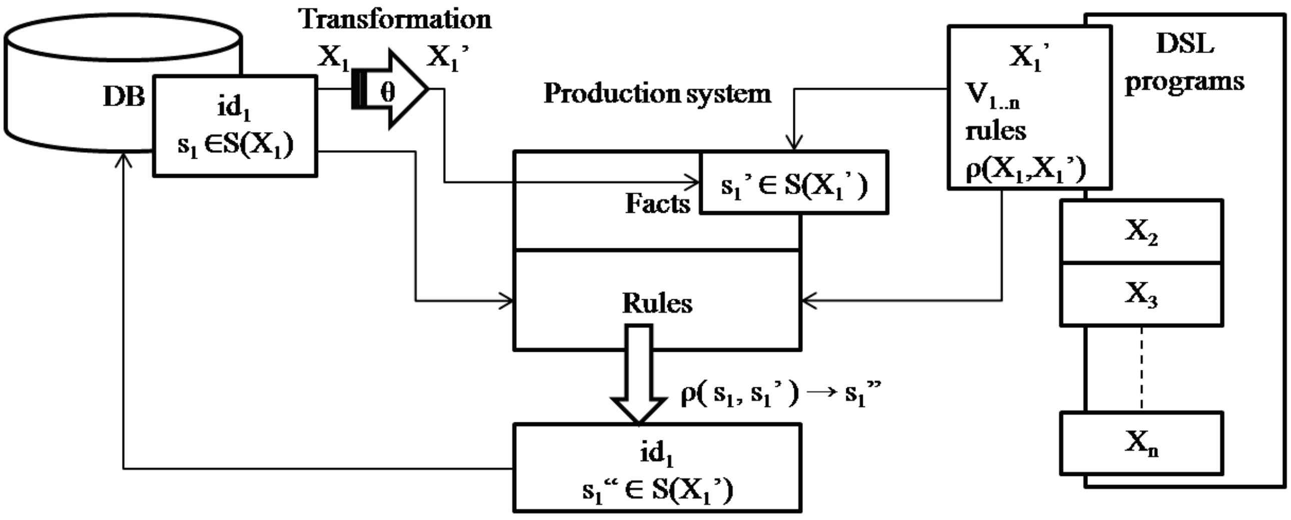


Figure 1.Reaction of the framework to ontology class evolution

The figure shows framework's database, the production system and a set of DSL programs that define a system's ontology classes $X_{1..n}$. The model description of the class X_1 is substituted with a newer version X_1'. The database contains an instance identified by id_1, that describes a certain state s_1 of X_1 's properties: $s_1 \in S(X_1)$, where $S(X_1)$ is the possible state space of X_1. When the newer version of the DSL program representing the class – X_1' is loaded in the production system, it loads a set of facts, corresponding to the newer set of properties ($V_{1..n}$) and the rules processing those. The new properties of the class X_1' do not correspond to the properties stored in the database and they cannot be asserted as values of the facts in the production system. To allow the assertion, the old facts are transformed in a trivial way, marked with θ from properties corresponding to the description of X_1, to properties, corresponding to the description of X_1', and are afterwards asserted in the production system:

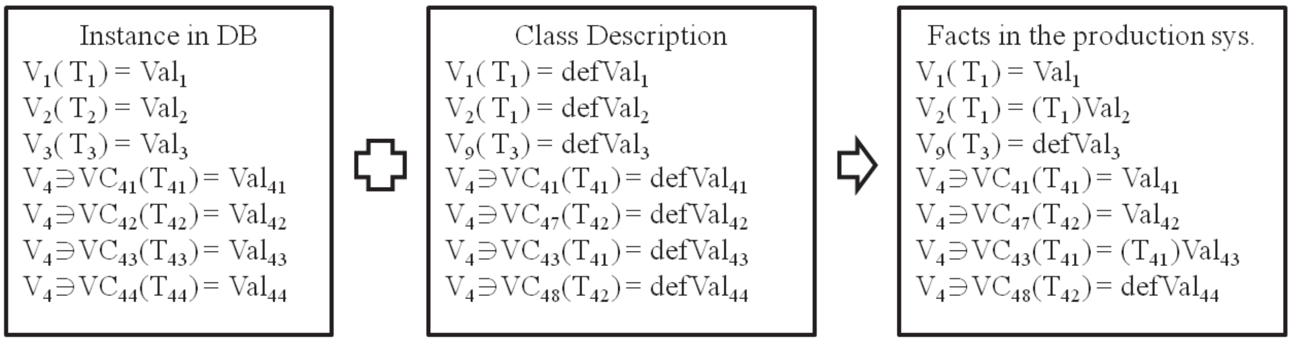
$$\theta(X_1', s_1) \mid s_1 \in S(X_1),\ s_1' \in S(X_1') \rightarrow s_1' \qquad \textbf{(1)}$$

After this transformation the rules in the production system are started with the set of facts corresponding to the state s_1'. There are additional rules to the ones specified in X_1' that are fired only when incosistency or incompleteness of the fact data is detected. This incompleteness corresponds to the difference of the attributes between the class descriptions of X_1 and X_1' – starting the rule $\rho(X_1, X_1')$. These rules' preconditions contain the trivially converted state s_1' and using custom funtions in the rules right hand side (RHS), load the database-stored state s_1. By applying a set of conversion operations on the stored state s_1, the rules achieve the new state s_1'', which adequately represents to the evolution of the schema described by the ontology class change from X_1 to X_1':

$$\rho(s_1,s_1') \mid s_1'' \in S(X_1') \rightarrow s_1'' \qquad \textbf{(2)}$$

1.2. Initial instance transformation

The instance transformation, described with expression (1) consists of finding identical identifiers of functional properties (i.e. properties with cardinality 1) and checking their types for accordance. Complex properties (ones with cardinality larger that 1) are compared by their identifiers and the type and name of the table columns described within them. For the initial phase, a trivial approach is chosen as compared to the cases reviewed in literature. Complex similarity analysis is not necessary, as additional rules are applied at later stage on the data. They can access the original properties and correct the crudely mapped ones accordingly. Figure 2 shows an instance stored in database and a new version of the class description for that instance, containing default values for the properties. The figure also shows an instance resulting from the trivial transformation of the original instance to the new property descriptions.



Instance in DB
$V_1(T_1) = Val_1$
$V_2(T_2) = Val_2$
$V_3(T_3) = Val_3$
$V_4 \ni VC_{41}(T_{41}) = Val_{41}$
$V_4 \ni VC_{42}(T_{42}) = Val_{42}$
$V_4 \ni VC_{43}(T_{43}) = Val_{43}$
$V_4 \ni VC_{44}(T_{44}) = Val_{44}$

Class Description
$V_1(T_1) = defVal_1$
$V_2(T_1) = defVal_2$
$V_9(T_3) = defVal_3$
$V_4 \ni VC_{41}(T_{41}) = defVal_{41}$
$V_4 \ni VC_{47}(T_{42}) = defVal_{42}$
$V_4 \ni VC_{43}(T_{41}) = defVal_{43}$
$V_4 \ni VC_{48}(T_{42}) = defVal_{44}$

Facts in the production sys.
$V_1(T_1) = Val_1$
$V_2(T_1) = (T_1)Val_2$
$V_9(T_3) = defVal_3$
$V_4 \ni VC_{41}(T_{41}) = Val_{41}$
$V_4 \ni VC_{47}(T_{42}) = Val_{42}$
$V_4 \ni VC_{43}(T_{41}) = (T_{41})Val_{43}$
$V_4 \ni VC_{48}(T_{42}) = defVal_{44}$

Figure 2. Trivial mapping between an instance and new version of its class

The property V_1 has type T_1 in both old and the new versions, so the database-stored value Val_1 is used in the resulting instance. When the type T_2 of the property V_2 is changed to another one – for example to T_1 in the new version of the class, the resulting instance is given a properly transformed value V_2 into the type T_1. When a property identifier is changed – like in the case when the property V_3 was renamed or substituted with V_9, the default value in the class description is used. Analysis of similarities between the two properties – V_3 and V_9 is avoided even if the two properties have the same type – T_3. If they are similar, the mapping can be performed in the rule ρ. Complex properties (VC) are associated to a single property identifier. In the example on figure 2 this is the identifier V_4. Every grid column, described in such a fact has a name (VC_{4i}) and type of the data inside it (T_{4i}). When the name and the type coincide as in VC_{41} – the database-stored value is used – Val_{41}. The same applies when column names are different, but their types are identical in the instance and the new version of the class. This is done to facilitate column renaming in the framework's tables. If the goal is to replace a certain column, this can be done in the transformation rule described in the next paragraph. When the name of a column coincide, but the types are different, as in VC_{43}, the database value is typecast accordingly. When neither the name, nor the type coincide, then the resulting instance's propery is given default values from the ones declared in the new class version. In this case, the framework inserts the same number of default values in the column as are present in the other columns of the property description to retain grid data consistency.

1.3. Rules for instance transformation

Using a set of rules to implement schema evolution is a known approach. Heflin and Hendler [5] present the SHOE (Simple HTML Ontology Extensions) language, which is a specialized extension to HTML for defining ontologies supporting versioning. Their language supports simple rules for connecting similar data in different ontology versions. Yu and Popa [14] discuss a way of database schema evolution representation through rules. They propose a single operation for evolution rather than the common approach of using many small incremental changes. Their approach, named Mapping-Based Representation (MBR) is based on rules, determining new fields in a new table for each changed value in the old table under some conditions, specified in a where clause. This allows them to adapt only a part of the data in a certain database table column to the new schema. Curino et al. [3] present the PRISM system, used for facilitating database schema evolution. They use Schema Modification Operators (SMO) that represent atomic schema changes and expand them with functionalities for converting types and semantic. The rules in SMO, besides being used for schema changes, are also used for automatic modification of the SQL queries to suit new definitions. Plessers et al. [10] present the Change Definition Language (CDL), that allows definition of structural change rules in ontological data. The expressions defined in CDL are parsed and converted to ontology definition and query languages like RDQL (RDF query language) and applied on the modified ontology.

The solution for ontology structure evolution presented in current publication uses lazy updates, applied on ontology class instances using production system rules. These rules have the full capabilities of the production system language and can be more complex than the solutions presented in the reviewed literature that use specialized languages or language extensions. This allows greater flexibility when transforming and recalculating new values if this is necessary. As an example we will review a "financial instrument" class evolution shown on figure 3. The field "Number of Months" is transformed to two new fields – "Maturity Date" and "Frequency". Proper schema transformation cannot be performed without calculating maturity date using the begin date and number of months. This is possible only if the transformation rules have the possibility to use procedural-type instructions as is the case for example with CLIPS expert system tool used in our implementation.

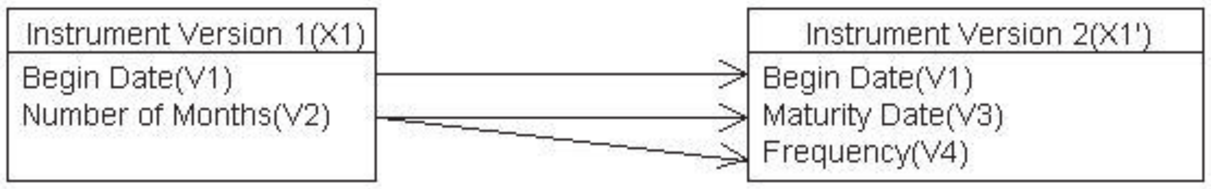


Figure 3. Nontrivial instance transformation

The class "Instrument Version 1" is denoted with X_1, and its instance in the database with s_1. The properties, all of which have cardinality of one, are denoted as follows: "Begin Date" is V_1, "Number of Months" – V_2, "End Date" – V_3 and "Frequency" – V_4. The goal of the transformation is to achieve the instance s_1" of the class X_1' ("Instrument Version 2"), that would represent the same data but correspond to the new schema. Using datatype annotation extensions of ALC described in [6], the following declarations are valid from the given definitions:

$X_1 \equiv =_1V_1 \cap V_1.\mathrm{real}[\mathrm{date}] \cap =_1V_2 \cap V_2.\mathrm{real}[\mathrm{int} \wedge \geq 1]$
$X_1' \equiv =_1V_1 \cap V_1.\mathrm{real}[\mathrm{date}] \cap =_1V_3 \cap V_3.\mathrm{real}[\mathrm{date}] \cap =_1V_4 \cap V_4.\mathrm{Frequency}$ **(3)**
$\mathrm{Frequency} \equiv \{$ Daily, Weekly, Monthly, Semianual, Annual $\}$

The trivial transformation of the instance s_1 into the intermediate instance s_1' gives the following results:

$\theta\ (X_1', V_1(s_1, a)) \rightarrow V_1(s_1', a)$
$\theta\ (X_1', V_2(s_1, b)) \rightarrow \perp$ **(4)**
$\theta\ (X_1', s_1) \rightarrow V_3(s_1', def_{V3})$
$\theta\ (X_1', s_1) \rightarrow V_4(s_1', def_{V4})$

, where a and b are constants corresponding to the domains of the properties V_1 and V_2 – i.e. a date and an integer larger than 1 and def_{V3} and def_{V4} are default values corresponding to the properties V_3 and V_4. After the trivial transformation, the temporary instance s_1' has one valid value – the begin date and two default values – end date and frequency. Also, the number of months from the database stored instance s_1 is ignored due to schema change. To achieve proper transformation, the following rule is added:

$\rho(V_1(s_1', a), V_2(s_1, b)) \rightarrow V_3(s_1', c) \cap V_4(s_1', \mathrm{Monthly})$ **(5)**

The rule uses the begin date from s_1' and the number of months from the database instance s_1, to calculate the value c of the property “End Date” (V_3) and sets the value “Monthly” to the property “Frequency” - V_4. The rule can be implemented with the following example CLIPS rule defined inside “Instrument”’s class DSL program:

```
(defrule VersionTransformation
        (InstanceIdentifier (str-value ?instanceId))
        (BeginDate (value ?startDate))
        (EndDate (value 0.0))
        (Frequency (value -1))
=>                                                                    (6)
        (if  (and (<> ?startDate 0.0)  (> (str-length ?instanceId) 0) )  then
        (bind ?months (GetFieldDB ?instanceId “Instrument” “NumberOfMonths”))
        (bind ?endDate (AddMonths ?startDate ?months))
        (assert (EndDate (value ?ednDate)))
        (assert (Frequency (value 2)))
        )
)
```

The rule has a precondition that requires the facts “End Date” and “Frequency” to have invalid values – the ones set by default. It associates the value slots of the facts “Begin Date” and “Instance Identifier” to the wildcards ?startDate and ?instanceId. The rule’s right hand side checks if the currently calculated instance of the class „Instrument” is transformed or newly created by comparing the values of the two wildcards with their corresponding default values. If they are not the default values, this means that the instance is transformed and the rule continues operation. It uses a core-supplied custom function “GetFieldDB” to load the value of the property “Number of Months” from the database. The proper instance is identified using its identifier and class. The function accesses the previous version of the instance to be transformed. After that, the rule uses a function “AddMonts” to add the loaded value to the begin date and calculate the end date of the instrument at hand. The custom functions are added to the production system by the framework core. Our example implementation has many such custom functions used for proper information system generation and faster calculations. In the end of the example code (6), two new facts are asserted in the production system knowledge base – containing the proper values for the facts “End Date” and “Frequency”. After performing both steps – the trivial transformation and the rule transformation, the new instance s_1” represents most accurately the data stored in initial instance s_1 and it can be stored back in the database. The described mechanism is suitable for migrating data between numerous versions because the preconditions of each version’s transformation rule can detect the proper database stored version.

2. Updating a group of instances to newer versions

The proposed approach allows also updating a group of database stored instances to their new schema descriptions. This is necessary when a change of properties of a certain class causes changes of properties of other connected ontology classes that also need to be updated. As an example, figure 4 shows a financial portfolio that references instruments from different versions in its instrument list property. To calculate its net present value (NPV) it has to transform all instances of the connected instruments to the last version – numbered 3 that contains a field “NPV”, collect the data from these fields and summarise them.

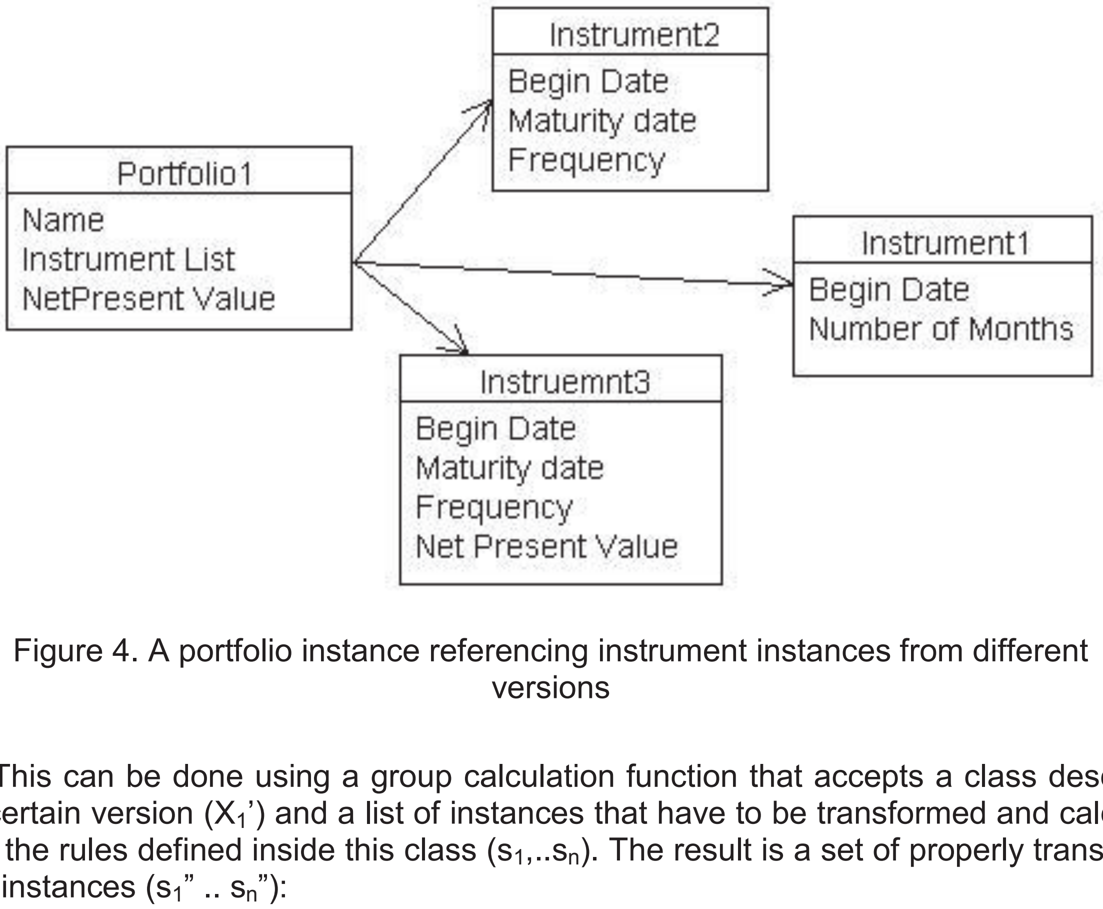


Figure 4. A portfolio instance referencing instrument instances from different versions

This can be done using a group calculation function that accepts a class description with certain version (X_1’) and a list of instances that have to be transformed and calculated using the rules defined inside this class (s_1,..s_n). The result is a set of properly transformed class instances (s_1” .. s_n”):

$$\begin{aligned} &\varphi(X_1',\{s_1, ..., s_n\})\rightarrow\{s_1'',...,s_n''\}: \\ &\qquad \rho(s_1,\theta(X_1',s_1))\rightarrow s_1'' \\ &\qquad \rho(s_2,\theta(X_1',s_2))\rightarrow s_2'' \qquad\qquad \textbf{(7)} \\ &\qquad ... \end{aligned}$$

$\rho(s_n,\theta(X_1',s_n)))\rightarrow s_n''$

In the example form figure 4, a calculation rule inside the DSL program of portfolio_1 calls the function $\varphi(X_3,\{\text{instrument}_1,\text{instrument}_2)\}$ and thus causes transformation of the instances instrument_1 and instrument_2. After running the transformation procedure, it can extract the NPV values from those instances using a function like the already presented "GetFieldDB" and summarize the values in its "Net Present Value" property. If necessary, the function φ can be called from the transformation rules of the individual instances - ρ, which can cause the update of instances of classes from the whole ontology. Of course, there is risk that with such recursive calls, an endless loop calculation could occur. The domain specific language programmers are responsible for avoiding such situations by preventing mutual dependencies in the ontology classes. If such dependencies cannot be avoided they should be very careful with using the group calculation function.

CONCLUSIONS AND FUTURE WORK

The paper reviews the way software versions are maintained in a framework for building ontology based information systems. The information systems are defined by DSL programs – one per ontology class. The transition to newer version is done by substituting these programs and updating the data about class instances already stored in the database. The instance update is reduced to applying the changes in terminological component (TBox) of an ontology on its assertional component (ABox). The proposed approach uses two-phase adaptation of the stored class instances – a trivial transformation followed by a tranformation through rules. The use of rules achieves optimal correspondence between old and new versions because they use complex operations during transformation. The specific structure of the information systems built by the framework – the fact that their business logic is also coded with rules - facilitates the use of rules in the described way. The version transition rules described in the publication are just a few among all the rules in the DSL programs that represent every class in the information system and are replaced when changing versions. A way for automatic update of instances was reviewed that allows any used instances of classes to be updated on demand. The method described allows easy and unnoticable for the end user version transition in the framework for building ontology based information systems.

REFERENCES

[1] Agrawal R., Somani A., Xu Y. *Storage and Querying of E-Commerce Data*. In Proc. of the 27th Int. Conf. on Very Large Data Bases, 2001, pp.149-158.

[2] Conradi R., Westfechtel B. *Version models for software configuration management*, ACM Comput. Surv. 30(2), 1998, pp.232-282.

[3] Curino C., Moon H., Zaniolo C. Graceful database schema evolution: the PRISM workbench. Proc. VLDB, 2008, pp.761-772.

[4] Hartung M., Terwilliger J., Rahm, E. Recent advances in schema and ontology evolution, Schema Matching and Mapping. Springer-Verlag, 2011,pp.149-190.

[5] Heflin J., Hendler J. Dynamic Ontologies on the Web. In Proc. of the 17th national Conf. on AI and 12th Conf. on Innovative Applications of AI, AAAI Press,2000, pp.443-449.

[6] Motik B., Horrocks I. OWL Datatypes: Design and Implementation. In Proc. of the 7th Int. Conf. on The Semantic Web, 2008, pp.307-322.

[7] Nikolov S., Antonov A. Framework for building ontology-based dynamic applications. In Proc. of CompSysTech '10, 2010, pp.83-88.

[8] Noy N., Kunnatur S., Klein M., Musen, M. Tracking changes during ontology evolution. In 3rd International Semantic Web Conference, 2004, Hiroshima, Japan.

[9] Papastefanatos G. et al., Vassiliadis P., Simitsis A., Aggistalis K., Pechlivani F., Vassiliou Y. Language Extensions for the Automation of Database Schema Evolution.

ICEIS 2008.

[10] Plessers P., Troyer O., Casteleyn S. Understanding ontology evolution: A change detection approach. Web Semantics: Science, Services and Agents on the WWW, Vol.5(1), 2007, pp.39-49.

[11] Ra Y., Rundensteiner E. A transparent schema-evolution system based on object-oriented view technology. IEEE Trans. on Knowledge and Data Engineering, Vol.9(4), 1997, pp.600-624.

[12] Stojanovic L., Maedche A., Motik B., Stojanovic N. User-Driven Ontology Evolution Management. In Proc. 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 2002.

[13] Xuan D., Bellatreche L., Pierra G. A versioning management model for ontology-based data warehouses. In Proc. 8th Int. Conf. on Data Warehousing and Know. Disc., 2006, pp.195-206.

[14] Yu C., Popa L. Semantic adaptation of schema mappings when schemas evolve. In Proc. 31st Int. Conf. on Very large data bases, 2005, pp.1006-1017.

[15] http://protege.stanford.edu/

[16] http://kaon2.semanticweb.org/

ABOUT THE AUTHOR

MSc. Student Samuil Nikolov, PhD Student, Department of Computer Systems, Technical University of Varna, Phone: +359 876 689 730, E-mail: samuil@eurorisksystems.com