Granular Ontologies and Graphical In-Place Querying

Janis Barzdins, Edgars Rencis and Agris Sostaks

Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia
{Janis.Barzdins, Edgars.Rencis, Agris.Sostaks}@lumii.lv

Abstract. The data ontologies in a form of UML Class diagrams are discussed in this paper. We call the data ontology granular, if its corresponding instance diagrams (data) can be divided into separate parts called slices. A typical example of granular ontologies is process ontologies, where slices are run-time instances of these processes. Based on the notion of granularity a graphical in-place query language is presented in this paper. The proposed language is easy to use by domain experts that are not IT specialists. Besides that it has a very efficient (linear) execution time for answering queries.

Keywords: Data ontologies, run-time instances, query language, graphical querying.

1 Introduction

While working with models, we have observed an interesting phenomenon – data can be often divided into separate parts naturally. These parts have their own semantics, which we would like to use while querying the model. If the division of the data ontology is well formalized, it is possible to develop a query language for the ontology that is both very efficiently executable and very convenient and easy-to-use for the end user being the domain expert, not an IT specialist.

In this paper we specify a set of ontologies, for which we can define a natural division in parts. We call these ontologies granular and define the granularity principles very formally in Section 2. After that, we describe the query language in Section 3, which we have developed for granular ontologies. Here we lay out the principles and primitives of the language, as well as define its time-efficiency.

We, however, understand that not all the real world ontologies fall into the class of granular ontologies. Therefore, in future we plan to extend the notion of ontology granularity a bit in order to widen the class of granular ontologies by extending the query language at the same time. The main objective that needs to be taken in mind in the process is that the query language must preserve its time-efficiency.

2 Granular Ontologies

In this paper we inspect data ontologies in a form of UML Class Diagrams. More precisely, we use only a subset of UML Class Diagrams containing classes, oriented associations, typed attributes and generalization.

From syntactic point of view our data ontology language is also a subset of the OWL (see the comparison in [1]). From the semantic point of view there is, however, a significant difference – while OWL uses the open-world semantics, we exploit the closed-world semantics. Our proposed data ontology language is a convenient mean for describing data of concrete domains, e.g., the structure of hospital registry. A very simple example of a data ontology describing study programs is seen in Fig. 1 (we use a traditional shorthand notation for associations, which are oriented in both directions).

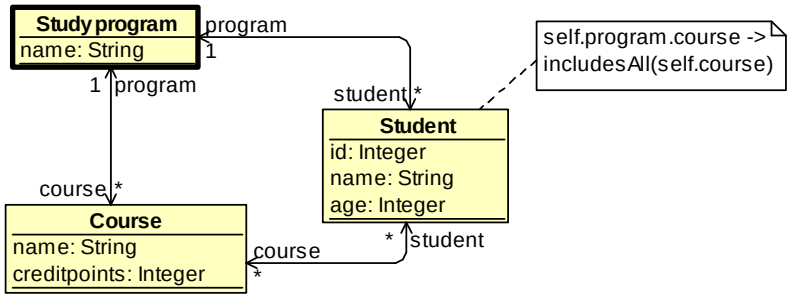

Fig. 1. The Study program ontology.

We will depict the concrete data of the ontology as *legal* instances of the corresponding class diagram. The specification of legality can be performed either only through multiplicities (which must always be satisfied), or additionally through OCL expressions (as in Fig. 1) or in any other way (even using the natural language).

Let us assume we have a data ontology in a form of UML Class Diagram $\mathcal{D}$ and a class A belonging to the ontology $\mathcal{D}$. Let us also assume we have some instance $\mathcal{G}$ of the diagram $\mathcal{D}$. $\mathcal{G}$ consists of two kinds of elements – class instances called *objects* and association instances called *links*. Since we only operate with oriented associations, also the links are oriented. Therefore we can perceive $\mathcal{G}$ also as an oriented graph. Let us now take an arbitrary instance x of the class A such that $x \in \mathcal{G}$ (in shorthand notation: $x \in A \cap \mathcal{G}$). We can now introduce a concept of a *slice respective to the object* x within instance $\mathcal{G}$ being the maximal subgraph of $\mathcal{G}$ (let us denote it with $S(x,\mathcal{G})$) such that $S(x,\mathcal{G})$ consists of the object x and all those objects that are reachable from x via edges.

When we inspect some data ontology $\mathcal{D}$, it always comes together with the set of its legal instances $\mathcal{U}_{\mathcal{D}}$. We will now call a class $A \in \mathcal{D}$ a *Master class*, iff the two following statements are satisfied:

1) $\forall \mathcal{G} \in \mathcal{U}_{\mathcal{D}} \forall x \in A \cap \mathcal{G} \forall y \in A \cap \mathcal{G} (x \neq y \Rightarrow S(x,\mathcal{G}) \cap S(y,\mathcal{G}) = \emptyset)$

2) $\forall \mathcal{G} \in \mathcal{U}_{\mathcal{D}} \bigcup_{x \in A \cap \mathcal{G}} S(x,\mathcal{G}) = \mathcal{G}$

The first statement states that all the slices respective to instances of the Master class are distinct, that is, they do not have common objects. The second statement states that these slices cover the whole instance $\mathcal{G}$.

There is only one Master class in the Study Program ontology seen in Fig. 1. (given the specified OCL constraint) – the class "Study program". Indeed, if we take an instance of the class "Study program", its respective slice covers all the courses of that program together with its students. Since the OCL constraint prohibits for any student to take course from a different program than he is attached to, it is clear that respective slices of instances of the class "Study program" are distinct thus dividing any legal instance of the class diagram into slices.

Data ontologies (together with the legality constraints), for which there exists a Master class, are called *granular* ontologies. We depict the Master class with a bold frame in granular ontologies as can be seen in Fig. 1 (in case there is more than one Master class in an ontology we just choose one). Further in this paper we inspect only granular ontologies.

The main objective of dividing the instance graph into slices is that thus we could form natural queries over the instance easily. Since the data are naturally divided into slices, we can formulate questions either within some concrete slice, or over a set of slices. For example, we can take one concrete slice specified by the name of the study program and count the sum of credit points of all courses of that program (e.g., the question "How much credit points are to be collected in the Computer Science Master study program?"). Another example – we can select a set of slices specified by the age of the students and see, in which study programs these students are assigned to (e.g., the query "Please, give me the list of all the programs, in which there are at least one student older than 30 years!"). The means for formulating such kind of queries and getting the results are described in the next Section.

It must be mentioned that the class of granular ontologies is relatively rich. We can see, how the division into slices becomes apparent in case of static class diagram seen in Fig. 1. However, the situation with division into slices is especially characteristic, if the data ontology describes some processes, and instances of the ontology are run-time instances (transactions) of those processes. Good example of such process is the shopping basket process widely used in the field of data mining. However, in this paper we will use another example as a base – clinical processes in a hospital. For describing such processes a special language MEDMOD is introduced in authors' paper [2].

Formally, the language MEDMOD is defined as a profile on UML Class diagrams according to Fig. 2 (OCL constraints are omitted here). An example clinical process describing the Emergency department management is seen in Fig. 3.

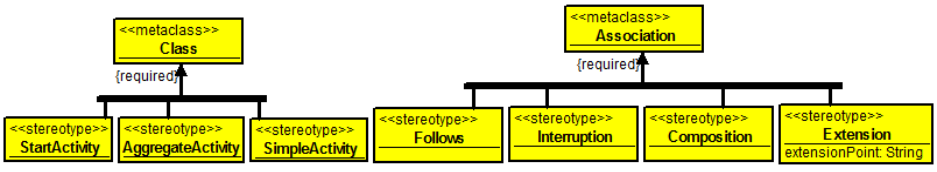

Fig. 2. The UML profile defining the MEDMOD language.

On the basis of Fig. 3 we will now shortly explain the used notations. As is described in the profile, Activities are divided into three categories. *Start Activity*, e.g., "Patient enters the hospital" (called also the Master Activity) is depicted with bolder frame in Fig. 3. *Aggregate Activities* (consisting of subactivities), e.g., "Clinical process in ward" are depicted with dashed frames (see Fig. 3). Simple activities are all the other activities, e.g., the activity "Doctor sets diagnosis" in Fig. 3. As is seen in Fig. 3, some activities are depicted with a multiple frame. That means that several instances (more than one) of these activities can appear in one slice.

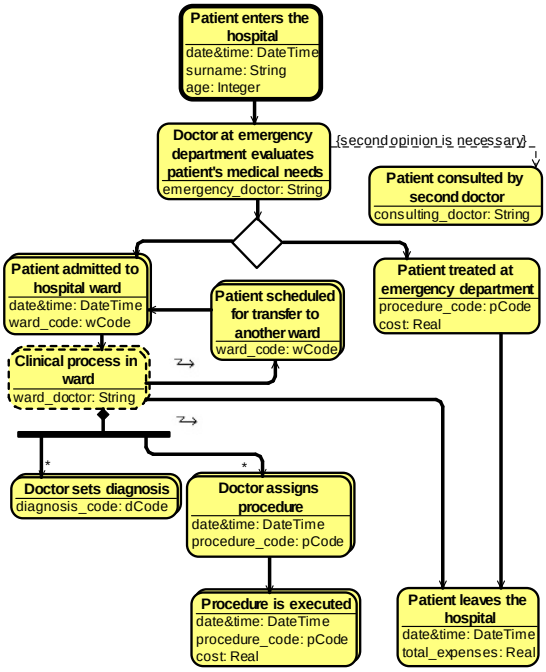

Fig. 3. An example of a MEDMOD process – the Emergency department management process.

Associations are divided into four categories:

1) Follows. This type of oriented relation can be established between two Activities A and B meaning that Activity B can only start after Activity A has ended. It is allowed for several Activities to follow the same Activity – the XOR semantics is implied in this case meaning that only one of those outgoing flows can be executed. We denote this situation by introducing a new diamond-shaped graphical element seen in Fig. 3.
2) Composition. A composition between two Activities can be established, if one Activity (called the Aggregate) semantically consists of one or more other Activities (called the Components).
3) Interruption. If there is an outgoing Interruption flow from the Aggregate Activity A to some Activity B, it means that the Activity A is suspended, when the flow is executed (i.e., when the Activity B needs to be started) meaning that it can no more create new Component instances (already created Component instances continues to execute normally).

4) Extension. Extension is an oriented relation between two Activities A and B meaning that Activity B can be called at some time during the execution of Activity A. The call is triggered, when some predefined condition occurs. The condition is described as an Extension point name and attached to the Extension.

The reason behind developing a new language was that the traditional process modeling languages have found a limited use in the hospital settings (see, e.g., [3], [4]). One of the reasons behind this delay has been the lack of clear definition of the sequence of activities that are carried out in clinical processes.

Since a MEDMOD diagram is formally a Class diagram according to the profile seen in Fig. 2, we can talk about instances of this class diagram, and we can investigate the notion of granularity of MEDMOD diagrams. The Master class comes out very naturally in this case, because the process diagram always has the starting action, which can serve as the Master class. The conclusion is that the ontology given by the MEDMOD language is granular.

Since the instance graph is again divided into slices (assuming we have formulated the instance legality criteria), we can query it either by specifying one concrete slice or several similar slices (e.g., "What is total expenses for the patient Wolf?"), or over a set of slices (e.g., "What is the average age of all patients treated by the doctor Stan Lee?").

The query language described in the next section is explained based on the MEDMOD example.

3 Query language

If an ontology is *granular* – its underlying instance graph can be divided into slices, then we can define simple and efficient means of querying the instance graph. In this Section we describe an ontology based graphical in-place query language that is easy to use even by non-IT specialists and the result of a query can be retrieved in the linear time $O(n)$ where n is the number of objects in the instance graph.

Since instance graph has been divided into slices accordingly to some granular ontology, questions can be asked accordingly to that ontology. Building a query has two main activities – **filtering** and **retrieving answers**. Filtering, actually, is setting simple constraints on objects. Constraints can be set on any attribute of any class in the ontology. Once a constraint has been set, the instance graph is reduced to those *slices***,** which contains at least one object that meets the constraint. Let's call it *the filtered instance graph*. We allow to retrieve answers for two types of questions, the first has an answer as a single number, e.g., *"How much did the Dr. Jekyll's patients cost?"* the second has an answer as a list of objects, e.g., *"Which patients with Pneumonia had no X-ray?"*.

Very important aspect of the query language is that its concrete syntax is based on the data ontologies language used to specify the ontology. An example of a query based on the MEDMOD language is given in the Fig. 4. The real-world examples, of course, are not as tiny as the given example - just 3 patients. An average hospital in Latvia (500 beds hospital) treats about 30 000 patients per year [5]. In order to better understand the query language we give an insight in the process of building queries.

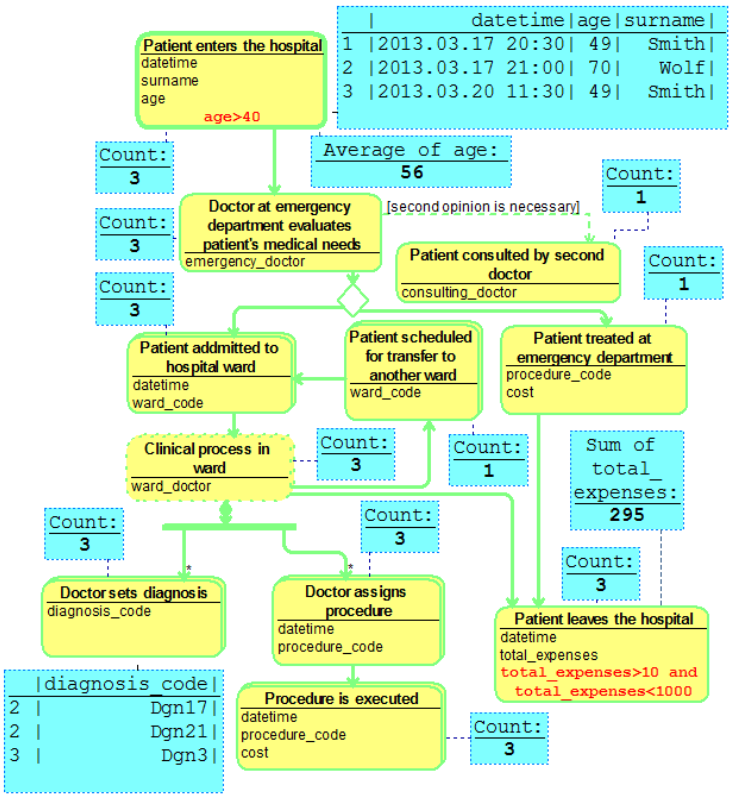

Fig. 4. An example of query based on MEDMOD – the emergency department management.

Let's assume that we have obtained the instance graph conforming to the given ontology (MEDMOD diagram). We leave behind the problem of getting data from hospital's information system. The person in interest (e.g., physician or manager) starts with a query diagram that is based on the given MEDMOD diagram – the query diagram has the same layout and elements as the MEDMOD diagram. It describes the familiar for the physician process of the emergency department management in the hospital. By default the query diagram contains boxes indicating the number of objects of each class in the instance graph. (See Fig. 4, boxes labeled *Count*). These are answers to simple questions like *"How many patients have been treated at emergency department?"* or *"How many procedures have been executed?"* It should be noted that every answer (result) is depicted as a box in the query diagram. Thus ontology, constraints, results - everything can be seen graphically *in-place* - in the same diagram. The same principle is used by spreadsheet applications – the user can make changes in any cell of the spreadsheet and observe the immediate effects on calculated values. In contrast most of query languages, e.g., SPARQL or SQL, have separate representations for data model, query and data. Now one can start filtering data by pointing to a class in the diagram and selecting an attribute. Simple constraints on attribute's values can be set – comparisons like *equals*, *greater than*, *less than*, etc., can be made to the constants of appropriate type. Following the

simplicity of spreadsheet applications, no more than two constraints (comparison operations) are allowed on each attribute. Both constraints may be mandatory (logical AND), or at least one of the constraints must be met (logical OR).

When a constraint on an attribute has been set, the instance graph is filtered and all answers (result boxes) in the diagram are reevaluated and all boxes refreshed. Thus the dynamic response to each step in construction of the complete query allows the physician to see immediate reaction to every action. It shortens the learning curve greatly and reduces the number of errors – they can be recognized much earlier. This effect is called *direct manipulation* interaction mechanism [6].

As it was mentioned earlier, all answers were depicted as boxes in the diagram. At any moment these boxes can be removed and new boxes can be added. Possible single number answers are: *Number of objects of given type in the filtered instance graph, Sum of values of given attribute in the filtered instance graph, Average of values of given attribute in the filtered instance graph*. The only allowed answer that is not a single number is *the list of objects (with attribute values) of given type in the filtered instance graph*. (See Fig. 4 for all types of answers).

Let's define the constraint, the query and the answer more formally. Assume that we have a granular ontology $\mathcal{D}$ which consists of classes which in turn contains attributes. Since the ontology $\mathcal{D}$ is granular, there exists some Master class $A \in \mathcal{D}$. Before one can query the instance graph $\mathcal{G}$, it must be divided into slices respective to objects of class A. Thus the queries must be executed over set of non-overlapping slices $\mathcal{S}$.

$$\mathcal{S} = \{ s \mid s = S(x, \mathcal{G}) \,\&\, x \in A \}$$

Slices consist of objects with associated key-value lists, where keys are attribute names and values are attribute values. The ontology determines possible attributes and their range of values (type) for objects of given class.

Let a be an attribute of some class $\mathcal{B} \in \mathcal{D}$ and let $T \in \mathcal{D}$ be the type of the attribute a. Then **constraint on attribute a** is the following Boolean expression:

1) One of the simple comparisons – $a > const$, $a = const$, $a < const$, where $const \in T$;
2) Conjunction (*and*) or disjunction (*or*) of two simple comparisons, e.g., "$a < 10$ *and* $a > 5$".

Such constraint can be checked on an object of class $\mathcal{B}$ in a time that consists of time that is needed to locate the value of the given attribute in the object's list of attribute values and time that is needed to do actual comparison and logical operations. Thus, the total time needed to check a constraint on a single object depends only on the size of the given ontology and implementation (coding) of objects. Therefore for each ontology and its implementation there exists such constant $\mathcal{C}$, that a single constraint can be checked on a single object in time less than $\mathcal{C}$.

As it was mentioned before, the physician is allowed to set just one constraint at once. After the constraint is set it is evaluated immediately. Let's define more precisely, what does it mean to evaluate a constraint c on attribute a of class $\mathcal{B}$ on the instance graph (set of slices $\mathcal{S}$) and obtain the filtered instance graph – the subset of $\mathcal{S}$.

The main idea is to go through all slices and check all objects in particular slice. If there is an object of the given type and the constraint c evaluates to *true* on that object, then the slice is added to the filtered instance graph. It is easy to see, that in the worst

case all objects in instance graph have to be checked to evaluate the constraint, but no more, because slices are non-overlapping. However, checking a single object does not require more time than the constant C, thus **the total time needed to evaluate a single constraint on the instance graph $\mathcal{G}$ is O(*n*), where *n* is the number of objects in $\mathcal{G}$.**

The complete **query** Q is the ordered set of constraints. The execution of the query starts with evaluation of the first constraint in the set and continues with gradual evaluation of next constraints on the result of the previous. As it was mentioned above, the typical number of patients treated in an average hospital in Latvia is 30000 per year. It would be the number of slices in the instance graph for the MEDMOD example. Our experience and initial experiments with query language show that last constraints in typical queries are evaluated on much smaller filtered instance graph comparing to the initial instance graph. It may allow us to predict that the execution of complex queries would be efficient for instance graphs even larger than abovementioned 30000 slices.

Now we can define more precisely, what **answers** (result boxes) can be retrieved. Once the filtered instance graph $\mathcal{FS}$ has been obtained, here are possible answers:

1) Number of instances of given class $\mathcal{B}$ in the filtered instance graph $\mathcal{FS}$

2) Sum of values of given attribute *attr* (in class $\mathcal{B}$) in the filtered instance graph $\mathcal{FS}$

3) Average of values of given attribute *attr* (in class $\mathcal{B}$) in filtered instance graph $\mathcal{FS}$

4) List of objects of given class $\mathcal{B}$ in the filtered instance graph $\mathcal{FS}$

Just like in case of constraints, also retrieving an answer does require a single inspection of an object in the instance graph. Thus, the total time to retrieve an answer on the instance graph $\mathcal{G}$ is *O(n)*, where *n* is the number of objects in $\mathcal{G}$. It should be noted that the query language may be extended without loss of efficiency by other means that also can be evaluated in the linear time, e.g., retrieving *average number of instances of given type per slice*, filtering *slices by number of instances of given type*, however we do not describe them all because of limitations of space.

To sum up, the main advantages of the proposed query language are:

- The view on data through "glasses" of familiar ontology (e.g., everybody in the hospital should know, how does it work!);
- The simple and easy-to-perceive means of setting filtering conditions require no more expertise than using spreadsheet applications (like *MS Excel*);
- The dynamic response to each step in construction of the complete query – the doctor sees immediate reaction to every action. It shortens the learning curve greatly and encourages even non-experienced users to try this out;
- The efficiency of query execution. It is required the linear time regarding to the size of the instance graph to filter and retrieve answers.

4 Related Work

Graphical query languages have been interesting to the researchers as long as textual query languages exist. They have been developed as an attempt to fulfill the promises of query languages to give an easy-to-use means for ad-hoc data analysis, because in

practice the powerful query languages (like SQL) have not became the mainstream tools for non-IT users. Number of graphical (visual) query languages for relational databases emerged in the late 80-s of the previous century [7, 8, 9]. However at that time the implementation of graphical languages was an expensive and time-consuming, not even thinking of usability issues that came along the involving non-IT users. They tend to cover every feature of SQL and as the result of that we can name just few examples of graphical query building tools, like, query designer in Microsoft Access [10] that provides means to build SQL queries graphically.

At the same time the spreadsheet applications have been widely used by non-IT users. They allow dealing with data in tabular form (no relations). One of the reasons of the spreadsheet's success story is the usage simple concepts like cell, row, column, etc., coming from real paper-based documents. Another reason is the dynamic response on every action that takes place in the spreadsheet – user sees all changes in the document immediately, just like in the query language we propose in this paper.

Nowadays the graphical language workbenches [11, 12] allow building graphical languages quickly. Thus the merely forgotten question about building visual query languages is back on the timetable. Ontologies have become popular in recent years. Therefore, the attention has been shifted from relational databases and ER models to ontologies. Thus the query languages for ontologies have emerged and particularly the graphical query languages for ontologies [13, 14]. And once again, these languages focus on graphical representation of the query, try to cover all features of SPARQL and separate the representations of ontology, query and data.

5 Conclusions and Future Work

One of the main results of this paper is the notion of granular data ontologies. This notion is defined very formally in the paper. Based on the notion of granularity an in-place graphical query language is then introduced. It is partly tested on real end-users – doctors of a hospital. As the first experience has shown, the query language possesses two essential features:

1) it is easily perceptible, and it is therefore easy to use by domain experts that are not IT specialists;
2) it has very efficient (linear regarding to the size of an instance graph) execution time for retrieving answers to queries.

Many noticeable data ontologies turn out to be granular, which means an efficient query language can be developed for them. At the same time there are also lots of other ontologies, which are not granular, and that prohibits us to use our query language for them. One of the main directions of our future research is to specify another meaningful class of data ontologies, which are granular in a wider sense. We will therefore extend the notion of ontology granularity allowing one to use the efficient query language for this class of ontologies. The efficiency of the query language will be preserved, i.e. the time evaluation of the query execution will remain linear. Other future research directions include, but are not limited to the following:

1) To keep on improving the query language and to test it on a wider range of potential end-users;

2) To continue optimizing the language implementation in order to improve the time needed for retrieving answers to queries formed over data containing about 30000 hospital transactions (our goal is to get the answer in less than a second here);
3) To further investigate practical use-cases of our approach in other areas outside the context of a hospital.

Acknowledgments

This work has been partially supported by the European Regional Development Fund within the project Nr. 2010/0325/2DP/2.1.1.1.0/10/APIA/VIAA/109 and by the Latvian National Research Program Nr. 2 „Development of Innovative Multifunctional Materials, Signal Processing and Information Technologies for Competitive Science Intensive Products" within the project Nr. 5 „New Information Technologies Based on Ontologies and Model Transformations".

References

1. Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A.: UML Style Graphical Notation and Editor for OWL 2. P. Forbrig and H. Günther (eds.), Perspectives in Business Informatics Research, LNBIP, Vol. 64, Springer, p. 102-113, 2010.
2. Barzdins, J., Barzdins, J., Rencis, E., & Sostaks, A.: Modeling and query language for hospitals. Health Information Science, LNCS, Vol. 7798, Springer, pp. 113-124, 2013.
3. Müller, R., & Rogge-Solti, A.: BPMN for Healthcare Processes. In Proceedings of the 3rd Central-European Workshop on Services and Their Composition, ZEUS 2011, Karlsruhe, Germany, February 21--22, pp. 65-72, Karlsruhe: CEUR-WS.org., 2011.
4. Agt, H., Kutsche, R.D., & Wegeler, T.: Guidance for domain specific modeling in small and medium enterprises. Proceedings of the compilation of the co-located workshops on SPLASH '11 Workshops, 63, 2011.
5. Central Statistical Bureau of Latvia, http://www.csb.gov.lv
6. Shneiderman, B.: Direct manipulation: A Step beyond Programming Languages, IEEE Computer, 16, pp. 57-69, 1983.
7. Angelaccio, M., Catarci, T. and Santucci, G.: QBD*: A graphical query language with recursion. In Proc. Third Human Computer Interaction Conf., 1989.
8. Cruz, I.F., Mendelzon, A.O. and Wood, P.T.: A graphical query language supporting recursion. In Proc. ACM SIGMOD Conf. Management of Data, 1987.
9. Czejdo, B., Embley, D., Reddy, V. and Rusinkiewicz, M.: A visual query language for an E-R data model. In Proc. Int. Workshop Visual Languages, Rome, Italy, 1989.
10. Microsoft Access – Office.com, http://office.microsoft.com/access
11. Sproģis, A., Liepiņš, R., Bārzdiņš, J., Čerāns, K., Kozlovičs, S., Lāce, L., Rencis, E., Zariņš, A.: GRAF: a Graphical Tool Building Framework. Proceedings of the Tools and Consultancy Track. European Conference on Model-Driven Architecture Foundations and Applications, Paris, France, pp. 18-21, 2010.
12. Graphical Modeling Framework (GMF) Tooling, http://eclipse.org/gmf-tooling
13. Zviedris, M., Barzdins, G.: ViziQuer: A Tool to Explore and Query SPARQL Endpoints, The Semantic Web: Research and Applications, LNCS, Vol. 6644, pp. 441 – 445, 2011.
14. Fadhil, A., Haarslev, V.: OntoVQL: A graphical query language for OWL ontologies. In: International Workshop on Description Logics. (2007)