Towards a Seed Transformation Language and Its Implementation

Sergejs Rikacovs

Advisor: Janis Barzdins

University of Latvia, IMCS, 29 Raina blvd, Riga, Latvia
{Sergejs.Rikacovs,Janis.Barzdins}@lumii.lv

1 Introduction

During the last few years a notable number of model transformation languages have been developed [1, 2, 3], etc. The biggest part of these languages contains some very advanced language constructs (such as declarative patterns). As a consequence, a development of an efficient implementation for this kind of languages becomes quite challenging.

The main objective of our research is to design and implement a new model transformation language that could be used as a seed transformation language. This language has to satisfy the following requirements:

- it has to be simple (from the point of view of its definition);
- an efficient implementation of this language must exist (it should be possible to obtain this implementation with a relatively moderate effort);
- it should be possible to incorporate some advanced features into this language (like undo/redo facilities, access rights policies and so on);
- it should be expressive enough to be usable in practical model transformation tasks
- it should be well suited for a role of a base language in a process of implementing higher-level model transformation languages by bootstrapping method.

The main advantage of having such language is that when some new feature is implemented in the seed language all languages being implemented on top of it obtain this feature without extra efforts. Increased portability and retargetability of languages implemented on top of the seed language are also important.

It should be noted that to some extent similar approach is employed in the ATL implementation [2]. Nevertheless, ATL approach does not satisfy some of the above-mentioned requirements and is based on different principles.

2 Initial Results

A first version of such a seed transformation language called L0 was proposed by the author of this paper and his colleagues in [4, 5].

2.1 Main Ideas of L0

L0 is a low level procedural strongly typed textual model transformation language. This language contains minimal but sufficient constructs for model and metamodel processing and control flow facilities resembling those found in assembler-like languages. An L0 program consists of a global variable definition's part and an L0 subprogram definition's part. An elementary unit of an L0 subprogram is a **command**. All L0 commands can be divided into two categories: commands for model processing and commands for metamodel processing.

Commands for model processing are as follows:

- creation/deletion of instances (like `addObj`, `addLink`, `deleteObj`, `deleteLink`);
- getting/setting the value of an attribute of an object (`setAttr`);
- iteration through instances (`first`, `next`).

For metamodel processing, L0 provides the following types of commands:

- metamodel building commands (creation of classes, attributes, associations, generalization relationships, etc.);
- metamodel elements scanning commands (commands which allow to iterate through metamodel elements).

2.2 Main Principles of Implementation

Thanks to the fact that L0 is a rather low-level language it is not difficult to build an efficient compiler for it. In this section, we sketch the main principles of this compiler. More information on this can be found in [5].

An important element of any model transformation language implementation is efficiently implemented model and metamodel storage and manipulation facilities. So it is natural to reuse facilities provided by already existing in-memory metamodel-based data stores [6, 7, 8, 9]. The API of these repositories provides ways to manipulate metamodel and corresponding instances in a reasonably efficient ways. For example, [7] has the following group of functions:

- a set of functions for creation and deletion of metamodel elements, as well as iteration through them;
- model processing functions:
  - functions for creation and deletion of instances (objects and links) and functions for getting/setting the value of an attribute:
  - efficient instance traversal functions.

Similar functions (with some minor differences) can be found in EMF, MDR and JGralab.

When such data store has been selected, compiler building is quite straightforward because most L0 commands specific to model or metamodel processing has close counterparts in an API of a corresponding data store. As to L0 control flow facilities, they can be easily emulated with control flow facilities found in a general purpose programming language.

We have successfully implemented an L0 compiler to Java (using [6] and [8] as data stores) and to C++ (using [7] as a data store).

2.3 Some Usages of L0

As it was mentioned above, the main use case of L0 is to be a base language in a process of implementing higher-level model transformation languages by bootstrapping method. This idea was practically verified by creating a sequence of model transformation languages, where the first language is L0 and every next language is obtained by adding more and more advanced constructs to the previous one. The last language of this family – L3 – is of sufficiently high level and contains such advanced constructs as loops and patterns. The family of these languages is called the Lx family [10]. By using L3 as an intermediate language, even a very high-level language MOLA containing declarative pattern facility has been successfully implemented [11].

L0 is also used outside the domain of compiler development. A notable example of this kind of usage is GrTP where L0 is used for transformation specification in the context of tool building platform [12].

3 Current Research

A problem sooner or later faced by a tool developer employing model transformations is the undo/redo problem. One way of solving it is to incorporate undo/redo facilities directly into the model transformation language. More precisely, we define a new language L0` by adding three new commands to L0:

- `CreateCheckpoint` – establishes a checkpoint;
- `Undo` – rolls back all changes made after the last checkpoint was established;
- `Redo` – cancels previously executed undo.

There are several ways how to implement L0`. One quite obvious approach is to use a data store, which has a built-in support for undo (EMF for example). The problem with this approach stems from the fact that not every repository provides such built-in facilities. As a consequence, if at some point we start using a repository that has this kind of facilities, we may end up experiencing serious difficulties trying to migrate to some other repository.

Our choice is to implement L0` by encoding all the necessary information in a model itself. It means we will create a compiler from L0` to L0 being able to create a corresponding L0 program P for every L0` program P`. In P, all undo-specific commands will be compiled to a set of corresponding L0 commands.

Let us describe the main ideas of this compiler. To be able to roll back changes made by transformation, we need to store information about performed actions (assuming that undo/redo will be available only for actions performed on a model level). We also should take special care while compiling the following L0 commands found in L0` program: link creation/deletion, object creation/deletion, and changing the value of an attribute.

To store information about these actions, we complement a metamodel of a model, which is to be processed by L0` program P`, with a fragment shown in Figure 1.(This fragment does not depend on a concrete domain metamodel. All other classes being

present in a user metamodel will become subclasses of the class “Object”.)

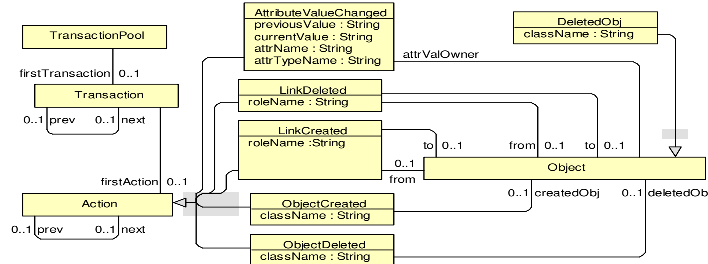

Fig. 1. Classes for supporting the undo facilities.

A compilation schema of L0` to L0 is sketched in Table 1.

Table 1. Schema of L0` compilation to L0.

L0` command	Description of a corresponding L0 code
`CreateCheckPoint`	Object of the class “Transaction” is created and is appended (via next/prev association) to the list of already existing transactions.
`addObj <pointerName>: <Classname>`	Object `o` of the type <ClassName> is instantiated. Object `a` of the type ObjectCreated is instantiated. Link of type createdObj between objects a and o is instantiated. Object `a` is appended to the list of already existing actions inside the current transaction.
`deleteObj <pointerName>`	Register attribute value changes. Delete instances of user defined associations in which this object takes part (see deleteLink). Create object `do` of the type DeletedObject. Create object `a` of the type ObjectDeleted. Repoint all instances of “undo specific” associations starting from object to be deleted to object `do`. Append object `a` to the list of already existing actions inside the current transaction.
`addLink <pointer1>. <assocname>. <pointer2>`	Link of the type <assocname> between <pointer1> and <pointer2> is instantiated. Object `a` of the type LinkCreated is instantiated. Link of the type fromObj between `a` and object pointed to by a <pointer1> is instantiated. Link of the type toObj between `a` and object pointed to by a <pointer2> is instantiated. Object `a` is appended to the list of already existing actions inside current transaction.
`deleteLink <pointer1>. <assocname>. <pointer2>`	Object `a` of the type LinkDeleted is instantiated. Link of the type fromObj between `a` and object pointed to by a <pointer1> is instantiated. Link of the type toObj between `a` and object pointed to by <pointer2> is instantiated. Link of the type <assocName> between objects pointed to by <pointer1> and <pointer2> respectively is deleted.

	Object a is appended to the list of already existing actions inside the current transaction.
setAttr <pointer>. <attrname> = <expr>	Object a of the type AttributeValueChanged is instantiated and populated with according values. Link of the type attrValOwner between a and object pointed to by <pointer> is instantiated. Object a is appended to the list of already existing actions inside the current transaction.
Undo	For each item in a list of actions of the current transaction (starting from the last one), perform an inverse action.
Redo	For each item in a list of actions of the current transaction (starting from the first one), perform a specified action.

The further works relating to the implementation of L0` will include validating of a proposed approach by means of performance benchmarking and supplementing L0` with possibilities to undo/redo actions done at the metamodel level.

4 Directions of Future Research

Let us consider building of graphical ontology browsers based on RDF data stores [13] – not so mainstream area of applications of model transformations to date. With ontology browsers, graphical tools providing a user-friendly way of ontology exploration is to be understood [14]. One way to implement this kind of tools is to do it by using facilities provided by GrTP [12].

A very important problem arising here is the problem of controlling access rights. This problem gets considerable amount of attention [15]. Nevertheless, it still lacks a satisfactory solution. It is believed that the "root of all evil" here is the fact that SPARQL is used as a tool for data access, because it is difficult to implement an efficient access rights aware SPARQL processor.

We are planning to study how this problem can be mitigated in case when RDF data are accessed through the facilities provided by transformation languages. It seems that if we access RDF database through a transformation language then (because of a different data access granularity) it is much easier to determine whether a user is allowed to access given datum or not.

To practically implement this idea, we are planning to extend L0` by introducing additional features – access right policies. Thus we will obtain a new language L0``, which would be able to control whether the given user has sufficient privileges to execute the given command. The next step would be to develop a compiler from L0`` to L0`.

5 Conclusion

The main goal of this thesis is to define a seed transformation language L0`…` containing a set of features necessary for implementation of higher-level model transforma-

tion languages by bootstrapping method. The second goal is to extend this seed language with some additional but practically important features such as undo/redo, access rights and so on.

References

1. OMG: MOF QVT Final Adopted Specification, URL: http://www.omg.org/docs/ptc/05-11-01.pdf
2. F.Jouault, I.Kurtev. Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica.
3. A.Kalnins, J. Barzdins, E.Celms. Model Transformation Language MOLA. – Proc. MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004) , Linkoeping, Sweden, pp.14-28
4. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, Model Transformation Languages and Their Implementation by Bootstrapping Method, Pillars of Computer Science., Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008, pp.130-145
5. S. Rikacovs, The base transformation language L0 and its implementation, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008, pp.75-102
6. Eclipse Modelling Framework URL : http://www.eclipse.org/emf/
7. J.Barzdins, G. Barzdins, R. Balodis, etc. Towards Semantic Latvia. Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006 , 2006), pp. 203-218
8. S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2006.
9. Metadata Repository (MDR). URL: http://mdr.netbeans.org
10. E. Rencis, Model transformation languages L1, L2, L3 and Their Implementation, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008, pp.103-139
11. A. Sostaks, A.Kalnins, The implementation of MOLA to L3 compiler, Scientific Papers, University of Latvia, "Computer Science and Information Technologies", 2008, pp.140-178
12. J. Barzdins, A. Zarins, K. Cerans, et al., *GrTP: Transformation Based Graphical Tool Building Platform*, MODELS 2007, Workshop on Model Driven Engineering Languages and Systems, 2007
13. Sesame. URL: http://www.openrdf.org
14. T. Berners-Lee, Y. Chen, L. Chilton , D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, D. Sheets: Tabulator: Exploring and Analyzing linked data on the Semantic Web. In: Proceedings of the 3rd Int. Semantic Web User Interaction Workshop, Athens, USA (2006)
15. P. Reddivari, T. Finin, A. Joshi: *Policy based access control for an RDF store*. Policy Management for the Web. (2005), pp.78-81