Towards Incremental Graph Transformation in Fujaba

[Position paper]

Gergely Varró
Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2.
H-1521 Budapest, Hungary
gervarro@cs.bme.hu

ABSTRACT

I discuss a technique for on-the-fly model transformations based on *incremental updates*. The essence of the technique is to keep track of all possible matchings of graph transformation rules, and update these matchings incrementally to exploit the fact that rules typically perform only local modifications to models. The proposal is planned to be implemented as a plug-in for the Fujaba graph transformation framework.

Keywords

graph transformation, graph pattern matching, incremental updates, Fujaba

1. INTRODUCTION

Model Driven Architecture. Recently, the Model Driven Architecture (MDA) of the Object Management Group (OMG) has become an interesting trend in software engineering. The main idea of the MDA framework is the use of models during the entire system design cycle. A major factor in the success of MDA is the development of industrial-strength models and various modeling languages. Several metamodeling approaches [2, 6, 19] have been developed to provide solid foundations for language engineering to allow system engineers to design a language for their own domain. As being the standard and visual object-oriented modeling language, UML obviously plays a key role in language design.

Transformation engineering in MDA. [20] However, the role of model transformations between modeling languages within MDA is as critical as the role of modeling languages themselves. As model transformations required by the MDA framework are supposed to be mainly developed by software engineers, precise yet intuitive notations are required for model transformation languages. QVT [16], an initiative of the OMG, aims at developing a standard for capturing Queries, Views and Transformations in MDA.

Incremental model transformations. During the design phase of the software engineering process, the system model may be modified several times, e.g., when correcting bugs, performing refinement steps, etc. When only a small portion of the model is modified, it is enough in general to re-execute a model transformation only on the part of the model that has actually been changed. This approach is called an *incremental (or on-the-fly) model transformation*.

The most typical example in a UML context is the incremental update of various views. A UML diagram shows one aspect of the system under design. If the system engineer modifies only one diagram, then modification may result in an inconsistent model. In order to maintain consistency, the design process should be supported by incremental model transformation, which updates all UML diagrams in a consistent way whenever any diagram changed. A related topic is discussed in [12], where consistency of logical and conceptual schemata of databases is maintained incrementally using traditional graph transformation techniques.

Incremental model transformations would also be advantageous for visual modeling languages. For instance, in [3], the authors discuss how the concrete syntax of a language can be generated from the abstract syntax by batch model transformations. However, incremental transformations would make this technique eligible to visual language editors, which require to automatically update the concrete syntax of the model according to the model-view-controller paradigm.

Fujaba as a model transformation tool. Fujaba, which is an Open Source UML CASE tool provides a rule-based visual programming language for manipulating the object structure based on the paradigm of graph transformation [18].

Traditionally, Fujaba has supported the specification of (and code generation from) the dynamic behavior of the system in the form of UML activity diagrams. Activity diagrams define the control flow of the methods and as such, they consist of activities (nodes) and transitions (edges). The role of transitions is to define temporal dependencies (i.e., execution order) between activities.

A graph transformation rule describes the behavior of a specific activity. A simplified version of UML collaboration diagrams (referred as story patterns) is used for specifying graph transformation rules. Activity diagrams that contain story patterns as activities are called *story diagrams* [8]. However, while Fujaba is considered to be one of the fastest graph transformation engines, there is still lack of support for incremental transformations.

Fujaba has been redesigned, and currently, it has a plug-in architecture. This new architecture still supports the basic code generation feature, but it additionally allows developers to easily add different functionalities while retaining full control over their contributions. As a consequence of this flexibility, several application

areas exist such as re-engineering [14], embedded real-time system design [1], education [15], etc.

Objectives. In the paper, I discuss the concepts of on-the-fly model transformation based on *incremental updates*. The essence of the technique is to keep track of all possible matchings of graph transformation rules, and update these matchings incrementally to exploit the fact that rules typically perform only local modifications to models. I plan to implement such an incremental graph transformation engine using Rete-algorithms [9]. The engine is planned to be integrated into the Fujaba graph transformation framework as a plug-in.

2. MODEL TRANSFORMATION

Visual modeling languages are frequently described by a combination of metamodeling and graph transformation techniques [6, 19].

2.1 Metamodeling

The *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have *attributes* that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Finally, *associations* define connections between classes.

In the MOF terminology [17], a metamodel is defined visually in a UML class diagram notation. In practical terms, the class diagram that has been designed in Fujaba by system engineers will form the metamodel in this case.

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required. In case of Fujaba, the generated concrete system will form the instance model.

Example. A distributed mutual exclusion algorithm whose full specification can be found in [11] will serve as a running example throughout the paper. *Processes* try to access shared *resources* in this domain. One requirement from the algorithm is to allow access to each resource by at most one process at a time. This is fulfilled by using a token ring, which consists of processes connected by edges of type *next*. In the consecutive phases of the algorithm, a process may issue a *request* on a resource, the resource may eventually be *held by* a process and finally a process may *release* the resource. The right to access a resource is modeled by a *token*. The algorithm also contains a deadlock detection procedure, which has to track the processes that are *blocked*.

The metamodel (type graph) of the problem domain and a sample instance model are depicted in the left and right parts of Fig. 1, respectively. The instance model presents a situation with two processes that are linked to each other by edges of type *next*.

2.2 Graph transformation

Graph transformation [5, 18] provides a pattern and rule based manipulation of graph-based models. Each rule application transforms a graph by replacing a part of it by another graph.

A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left–hand side graph LHS, a right–hand side graph RHS, and negative application condition graphs NAC.

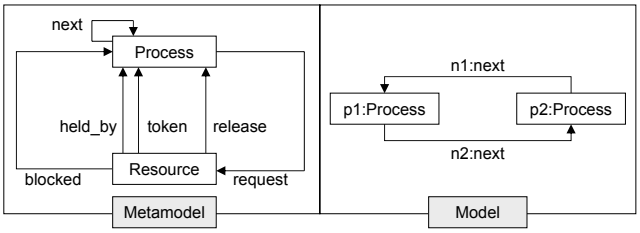

Figure 1: A sample metamodel and instance model

The *application* of r to an *host (instance) model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. The latter two steps form the so-called updating phase. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example. A sample rule of the distributed mutual exclusion algorithm (depicted in Fig. 2) simply inserts a new process between neighboring processes p1 and p2.

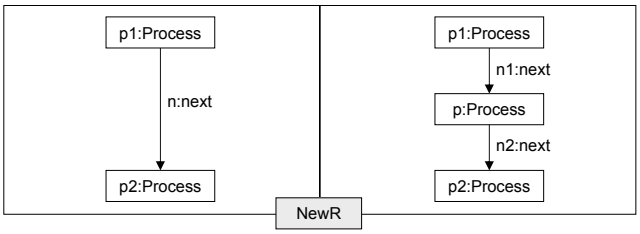

Figure 2: A sample transformation rule (newR)

2.3 Graph pattern matching

Typically, the most critical phase of a graph transformation step concerning the overall performance is graph pattern matching, i.e. to find a single (or all) occurrence(s) of a given LHS graph in a host model.

Current graph transformation engines use different sophisticated strategies in the graph pattern matching phase. These strategies can be grouped into two main categories.

- Algorithms based on *constraint satisfaction* (such as [13] in AGG [7], VIATRA [21]) interpret the graph elements of the pattern to be found as variables which should be instantiated by fulfilling the constraints imposed by the elements of the instance model.

- Algorithms based on *local searches* start from matching a single node and extending the matching to the neighboring nodes and edges. The graph pattern matching algorithm of PROGRES (with search plans [23]), Dörr's approach [4], and the object-oriented solution in FUJABA [8] fall in this category.

However, it is common in all these engines that they can be characterized as having a complex pattern matching phase followed by a simple modification phase and these phases are executed iteratively.

The main problem is that the information on previous match is lost, when a new rule application is started. As a consequence, the complex pattern matching phase has to be executed from scratch again and again. However, because of the local nature of modifications, it may be expected that the majority of matchings remain valid in consecutive steps. The same matchings are calculated several times, which seems to be a waste of resources in case of e.g., long transformation sequences.

3. INCREMENTAL UPDATES

In order to avoid recalculation of matchings, we proposed a technique based on *incremental updates* [22], for implementing efficient graph transformation engines designed especially for incremental (on-the-fly) model transformations. The basic idea in a graph transformation context is to store information on previous match and to keep track of modifications.

Several other solutions already exist for reducing the overhead of finding matches for LHS of rules as implemented in PROGRES [23]: (i) applying a graph transformation to all matches in the graph as one graph rewriting step (pseudo-parallel graph transformation), (ii) using incrementally computed derived attributes and relationships in LHS, and (iii) using rule parameters in graph transformations to pass computed knowledge about possible LHS matches from one rule to the next one.

After many years of research, different techniques based on the incremental updating idea have evolved and by now they are widely accepted and successfully used in several types of applications (e.g., relational databases, expert systems).

- In the area of relational databases, views may be updated incrementally. A database view is a query on a database that computes a relation whose value is not stored explicitly in the database, but it appears to the users of the database as if it were. However, in a group of methods, which is called by view materialization approach, the view is explicitly maintained as a stored relation [10]. Every time a base relation changes, the views that depend on it may need to be recomputed.

- In the area of rule-based expert systems, the Rete-algorithm (for more details see [9]) uses the idea of incremental pattern matching for facts. First a data-flow network is constructed based on the condition (*if*) parts of rules, which is basically a directed acyclic graph of a special structure. Initially, this network is fed by basic facts through its input channels. Compound facts are constituted of more elementary facts, thus they are the inputs of internal nodes in the network. If a fact reaches a terminal node, then the rule related to this specific node becomes applicable and assignments modifying the set of basic facts may be executed (according to the *then* part). Since every node keeps a record of its input facts, only modifications of these facts have to be tracked at each step.

Despite these results, (quite surprisingly) no graph transformation tools exist that provide support for incremental transformations. In [22], we carried out some initial experiments, which used an off-the-shelf relational database to measure the performance of the incremental updating method compared to the traditional (from scratch) approach. However, it turned out the most relational databases do not support incremental view updates. Therefore, it seems to be necessary to develop a new incremental graph transformation engine from scratch.

In the current paper, I propose to build a graph transformation engine that uses the Rete-algorithm for implementing the incremental updating technique.

Now I sketch the basic structure of such an engine. A graph transformation rule can be viewed as a rule that has a condition (*if*) and an action (*then*) part. The condition part corresponds to the LHS of the graph transformation rule, while the action part consists of all the actions (delete, update, insert) that have to be executed in the updating phase. According to this mapping, we can build a data-flow network for each rule using the LHS. Nodes and edges of the LHS are mapped to input nodes, while the whole LHS will correspond to a terminal node. The data-flow network may also have some internal nodes, which are basically subgraphs of the LHS. After this network building phase we will have as many data-flow (Rete) networks as many rules we originally have. Then these networks are merged by the Rete-algorithm in order to decrease the number of nodes.

Note that the nodes and edges of the metamodel and the actual instance model will appear as input nodes and basic facts assigned to the corresponding input nodes, respectively. Basic facts flow through the network and constitute more and more compound facts as they progress. When a compound fact reaches a terminal node, then the corresponding graph transformation rule becomes applicable, and the updating phase can be executed. This phase actually modifies the active set of basic facts assigned to input nodes.

In an ideal case, such an incremental graph transformation engine should be available as a plug-in for many graph transformation tools (thus being independent of them). However, since the interfaces of graph transformation tools are not (yet) standardized I plan to integrate the incremental engine as a transformation plug-in of Fujaba. This would provide an *alternate graph transformation engine* tailored especially to incremental model transformations (possibly defined by triple graph grammar rules). However, no modifications are required to the base system of Fujaba.

Example. In order to sketch the idea of incremental updates, let us consider that rule newR (depicted in Fig. 2) is trying to be applied to the instance model of Fig. 1. The pattern matching phase selects two valid subgraphs of the instance model, on which the rule is applicable. The transformation engine then executes the updating phase resulting in a model that contains 3 processes that are stringed on a chain consisting of 3 edges of type *next*.

Up to this point, both traditional and incremental approaches do the same. But when the pattern matching phase of the following rule application is executed, the traditional approach recalculates valid matchings from scratch, while the incremental method only has to delete invalid matchings and generate new ones. The first method should examine all the *next* edges appearing in the instance model, which may contain an arbitrary number of *next* edges. However, in case of the incremental technique, it is enough to examine only such *next* edges that are actually removed or created in the previous step. The number of such edges are always three in this example regardless of the size of the instance model.

Naturally, in case of dozens (hundreds) of transformation rules, a single application of a rule might need to recalculate the matching of several rules therefore, there is certainly a trade-off between a cheap pattern matching phase and a more complex update phase. I also intend to carry out experiments to assess this trade-off between traditional (batch or programmed) and incremental transformations.

4. CONCLUSIONS

In this paper, I discussed the necessity of incremental model transformations in the context of the Model Driven Architecture (transformation-based derivation of concrete syntax from abstract syntax in visual modeling languages, consistent and on-the-fly update of UML diagrams, etc.). I discussed the concepts of incremental model transformations based on the paradigm of graph transformation. I plan to implement such an engine using Rete-algorithms and integrate it into Fujaba as a plug-in. Furthermore, I would like to investigate the applicability of the incremental approach to various model transformation techniques (including triple graph grammars).

5. ACKNOWLEDGMENT

I am very much grateful to Dániel Varró and Andy Schürr for giving valuable comments and hints on incremental updates strategies and/or the paper itself.

6. REFERENCES

[1] S. Burmester and H. Giese. The Fujaba real-time statechart plugin. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[2] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation semantics for UML. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[3] P. Domokos and D. Varró. An open visualization framework for metamodel-based modeling languages. In *Proc. GraBaTs 2002, International Workshop on Graph-Based Tools*, volume 72 of *ENTCS*, pages 78–87, Barcelona, Spain, October 7–8 2002. Elsevier.

[4] H. Dörr. *Efficient Graph Rewriting and Its Implementation*, volume 922 of *LNCS*. Springer-Verlag, 1995.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[6] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[7] C. Ermel, M. Rudolf, and G. Taentzer. *In [5]*, chapter The AGG-Approach: Language and Tool Environment, pages 551–603. World Scientific, 1999.

[8] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*. Springer Verlag, 1998.

[9] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 1982.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 1995.

[11] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE'98*, volume 1382 of *LNCS*, pages 138–153. Springer-Verlag, 1998.

[12] J. H. Jahnke, W. Schäfer, J. P. Wadsack, and A. Zündorf. Supporting iterations in exploratory database reengineering processes. *Science of Computer Programming*, 45(2-3):99–136, 2002.

[13] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.

[14] J. Niere. Using learning toward automatic reengineering. In *Proc. of the 2nd International Workshop on Living with Inconsistency*, 2001.

[15] J. Niere and C. Schulte. Thinking in object structures: Teaching modelling in secondary schools. In *Proceedings of the ECOOP Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts*, 2002.

[16] Object Management Group. *QVT: Request for Proposal for Queries, Views and Transformations*.

[17] Object Management Group. *Meta Object Facility Version 2.0*, April 2003.

[18] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.

[19] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[20] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Proc. UML 2004: 7th International Conference on the Unified Modeling Language*, 2004. In press.

[21] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[22] G. Varró and D. Varró. Graph transformation with incremental updates. In *Proc. 4th Int. Workshop on Graph Transformation and Visual Modeling Techniques*, 2004.

[23] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, 1996.