Identification and Application of a Model Transformation Design Pattern

Hüseyin Ergin
hergin@crimson.ua.edu

Eugene Syriani
esyriani@cs.ua.edu

Department of Computer Science
University of Alabama
Tuscaloosa AL, U.S.A.

ABSTRACT

Model-driven engineering and model transformation have gained significant importance in recent years. However, most model transformation developments are handled without quality in mind. This paper presents a new model transformation design pattern which improves quality in model transformation. We discovered the pattern after solving a specific problem with two alternative model transformation designs. The improved solution turns out to be applicable to other problems and, from this recurrence, we generalize the solution to a design pattern.

Keywords

model transformation, quality, design patterns

1. INTRODUCTION

Model-driven engineering (MDE) [8] is considered a well-established software development approach that uses abstraction to bridge the gap between the problem and the software implementation. These abstractions are defined as models. Models are primary artifacts in MDE and are used to describe complex systems at multiple levels of abstraction, while capturing some of their essential properties. These levels of abstraction let domain experts describe and solve problems without depending on a specific platform or programming language.

Models are instances of a meta-model which defines the syntax of a modeling language. In MDE, the core development process consists of a series of transformations over models, called model transformation. Model transformations take as input and output a model according to their specification defined at the meta-model. Model transformation languages consist of two main components: *rules* and *scheduling*. Rules are the smallest units of a model transformation and are defined with pre-condition and post-condition patterns. The pre-condition pattern determines the applicability of a rule and is usually defined with a left-hand side (LHS) and optional negative application conditions (NACs). The LHS defines the pattern that must be found in the input model in order to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. The post-condition pattern determines the result of the application of the rule and is defined by a right-hand side (RHS) which must be satisfied after the rule is applied. The scheduling describes the order in which rules are executed.

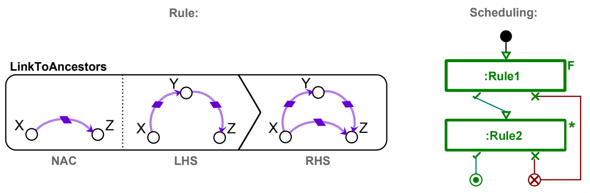

Figure 1: Sample MoTif rule and scheduling

To develop a model transformation, developers design the different rules and specify the scheduling. However, this design phase lacks of re-usability, which hampers the quality of model transformations. Therefore, there is a need for reusable, proven, and qualified structures in this phase. A design pattern encapsulates a proven solution to a recurring design problem [3]. As in the object-oriented world, design patterns help with the assessment of high quality model transformations. In this paper, we propose to solve the well-known lowest common ancestor (LCA) problem [2] using model transformation. For this purpose, we solve the problem using a naïve and an improved solution. We show that the latter improves the quality metrics of the model transformation with respect to efficiency criteria. Then, we identify two other problems that can be solved using an approach similar to the improved LCA solution. We therefore generalize the solution to a design pattern as it describes a solution for recurrent problems and increases the quality of the model transformation that implements it.

In this paper, we are using MoTif [9] as our model transformation language. MoTif is a rule-based model transformation language with explicit rule sheduling defined as a control flow. Explicit rule scheduling lets designers define the control flow of their rules. A sample rule and scheduling from MoTif is depicted in Figure 1. This rule can be read as “if there is a link from node X to node Y and a link from node Y to node Z, then there must be a link from node X to node Z, unless there is already one.” In the scheduling

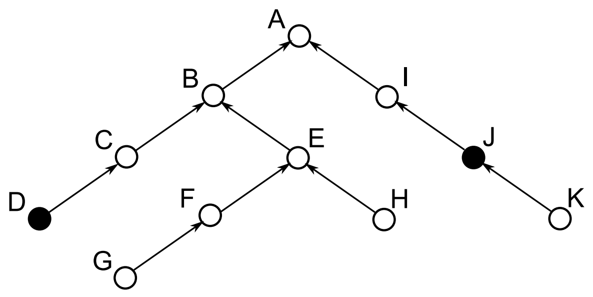

Figure 2: Tree instance for LCA problem

part, each rule is represented by a rule block having three ports. Conceptually, a rule receives models via the input port at the top. If the rule is successfully applied (meaning the post-condition is fulfilled, the resulting model is output from the success port at the bottom right. Otherwise, the model does not satisfy the pre-condition (no occurrence of the LHS without the NAC is found in the model) and the original model is output from the fail port at the bottom left.

In Section 2, we describe the naïve and improved solutions of the LCA problem and report some of the quality metrics. In Section 3, we introduce the solutions of equivalent resistance problem and Dijkstra's shortest path algorithm. Section 4 presents the generalized solution as a model transformation design pattern. Section 5 explores existing work in model transformation design patterns and quality metrics. Finally, in Section 6, we discuss and conclude the study.

2. RUNNING EXAMPLE

LCA is a general problem in graph theory and is typically defined over a directed tree structure. Essentially, it attempts to find the lowest shared ancestor between two given input nodes of the tree. For example in Figure 2, the LCA of nodes D and J is node A. In this instance, one can compute the LCA of node D and node J to be node A.

2.1 Naïve Solution

Typically, solutions using model transformation approaches tend to take advantage of the declarativeness and non-determinism of rule-based systems. In the first solution we propose, we first create all ancestor links of every node as depicted by the first three rules in Figure 3. Then GetLCA rule checks if, given the two initial nodes (A and B), there is an ancestor node common to both nodes that do not have a successor that is also a common ancestor of the two nodes. The rules and scheduling of these rules are depicted in Figure 3. For this study, we have focused on three metrics: *the number of rule applications* counts how many times the rule is applied, *the size of the rule* counts the number of elements present in the patterns of each rule, and *the number of auxiliary elements created* counts the number of ancestor links created to compute the LCA.

To compute the metrics, we consider a tree with n nodes and hence $n-1$ edges. The LinkToSelf rule creates self-ancestor links for all nodes, to cover the trivial case, and is applied n times, once for every node in the tree. The LinkToParent rule creates ancestor links to the parents of each node and is applied $n-1$ times, once per edge. The LinkToAncestors rule creates ancestor links to all ancestors of each node, recursively. Therefore, the number of ancestor links is proportional to the depth of each node. The following equation gives the total number of ancestor links that need to be created, where k_i is the depth level of node i.

$$\sum_{i=1}^{n} k_i - 2 = O(n^2)$$

After all ancestor links are created, the GetLCA rule is applied only once and returns the LCA of the given input nodes if it exists. The NAC part of the GetLCA rule guarantees that the solution is the lowest one among other common ancestors. The metrics for the naïve solution are depicted in Table 1.

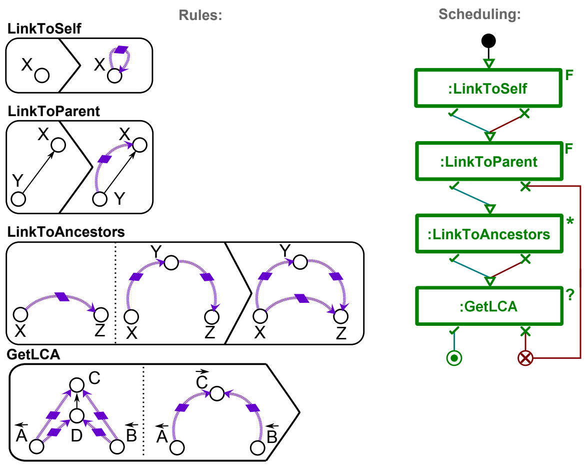

Figure 3: Rules for naïve solution

2.2 Improved Solution

In the improved solution, we use locality, focusing on only the given input nodes. We adopt an iterative approach. We start to create ancestor links one step at a time and, at each time, we check for a solution. The rules and scheduling of these rules are depicted in Figure 4.

The LinkToSelf rule creates self-ancestor links for the given input nodes only and therefore is applied twice. To acheieve that, we use the pivot feature in MoTif which forces the rule to be applied on bound or elements. That is, A and B are parametrized nodes bound to nodes from the input model at run-time. Then, the LinkToParent rule creates ancestor links to the parents of input nodes, which is applied twice. This results in an intermediate form of the tree instance, which may possibly solve the LCA task. Therefore, we apply the GetLCA rule and try to find the solution if it exists. If we cannot find a solution, we execute the LinkToAncestor rule and create one more level of ancestor links. Again, we use only the given input nodes. With only one more step, this rule takes the intermediate form closer to a solution. Then, we use the GetLCA rule to check again. These iterative steps continue until the GetLCA rule finds a solution or the LinkToAncestor rule fails by not making a contribution to the solution *i.e.,* if the root is reached and GetLCA fails. For the tree instance in Figure 2, the solution is found in three steps. Therefore, the GetLCA rule is applied four times and the LinkToAncestor rule is applied three times. In general, the given input nodes might be in different depth levels (k_1 and k_2 respectively). The ancestor link creation contin-

Table 1: Metrics for naïve and improved LCA solutions

Rules	Size of rules		# Rule Applications		# Auxiliary Elements	
	Naïve	*Improved*	*Naïve*	*Improved*	*Naïve*	*Improved*
LinkToSelf	3	3	n	2	$n-1$	2
LinkToParent	7	7	$n-1$	2	$O(n^2)$	$2n-2$
LinkToAncestors	14	14	$O(n^2)$	$2n-2$	$O(n^2)$	$2n-2$
GetLCA	14	14	1	n	0	0
Total	**38**	**38**	$\mathbf{O(n^2+2n)}$	$\mathbf{3n+2}$	$\mathbf{O(n^2+2n)}$	$\mathbf{2n+2}$

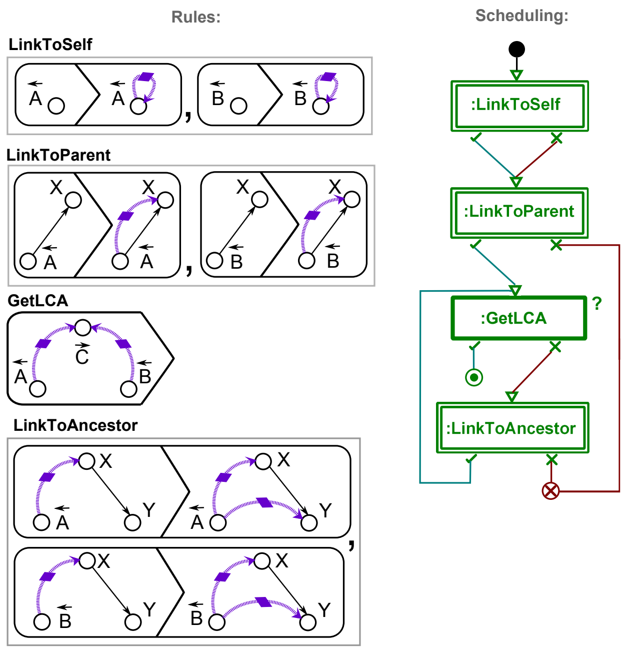

Figure 4: Rules for improved solution

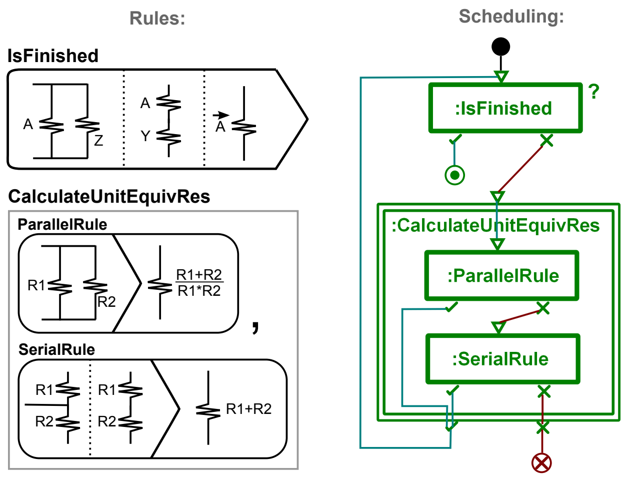

Figure 5: Rules for Equivalent Resistance Problem

ues up to the root node, so the maximum of depth levels is the number of iterations needed to find the solution. In the worst case, this depth can be n and we create $n-1$ ancestor links. Therefore, the LinkToAncestor rule is applied a total of $2(n-1)$ times for input nodes and the GetLCA rule is applied n times.

Metrics for the improved solution are also depicted in Table 1. One can clearly see the improvement by comparing the metric counts between naïve solution and improved solution. All three metrics are related to the efficiency quality criteria. Therefore, we can say the improved solution is more efficient than the naïve solution. We did not take the execution time of the model transformations because they are already proportional to the enumerated metrics.

3. SIMILAR PROBLEMS

In this section, we identify and solve two more problems from very different domains using model transformation.

3.1 Equivalent Resistance

In electrical circuits, the computation of the equivalent resistance of the whole circuit is a common task. Finding the equivalent resistance in a series of connected resistors is an interesting problem to apply our design pattern. In this case, the transformation takes as input an electrical circuit model with resistors connected both in serial and parallel. The rules are depicted in Figure 5. The IsFinished rule looks for resistors set in serial or parallel in the circuit. If the rule cannot find any more serial or parallel resistors, it will return the single resistor as the equivalent resistance. The CalculateUnitEquivalentResistance rule calculates equivalent resistance for only a set of serial and/or parallel resistors and directs the control flow to the IsFinished rule again depicting a loop.

3.2 Dijkstra's Algorithm for Shortest Path

Dijkstra's algorithm is a well-known graph search algorithm that returns the shortest path and length of this path between two nodes, source and target. The solution is provided in Figure 6. The input model is a directed and weighted tree. The VisitImmediateNeighbors rule initiates the algorithm by visiting the immediate neighbors of the source node. After a visit, each node is assigned with the weight of the path and is colored in red to represent that it is visited. The terminating criteria of the algorithm is visiting all nodes, which is ensured by the IsAllNodesVisited rule. If there are still unvisited nodes, then the VisitOneMoreHop rule is executed. The VisitOneMoreHop rule selects the smallest number of weighted nodes among visited ones and calculates the new weights for the unvisited neighbors of this node. After each node is visited, the target node will have the length of the shortest path as value and the path with purple marked arrows will be the shortest path.

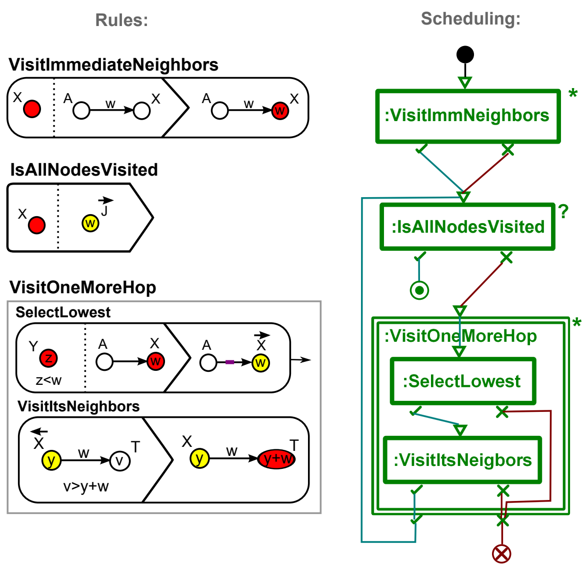

Figure 6: Rules for Dijkstra's Algorithm

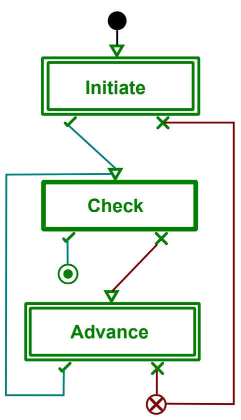

Figure 7: Generalization of the improved solution in three blocks

4. GENERALIZATION OF THE SOLUTION

The improved LCA, equivalent resistance, and Dijkstra's shortest path model transformation solutions look very alike. The structure is like a fixed-point iteration. In general there are three blocks. The first block initializes the input model with creation of some temporary elements and results in an intermediate form of the model (Initiate step). The initialization is optional (*e.g.,* Equivalent resistance problem in Section 3.1) but we have to include it in generalization. Then, a query verifies if a solution if found (Check step). Finally, if the query fails, the last block encodes one more step towards the solution (Advance step). The structure can also be seen as a while not loop in programming languages.

Gamma *et al.* [3] presented the design patterns they provided in a structured format. In their book, each design pattern has some essential elements that need to be described. These elements are *pattern name*, *problem*, *solution* and *consequences*. There are also some helper elements such as classification, applicability, structure, and participants. Following their format this study, we describe the newly identified model transformation design pattern as follows:

- **Pattern name:** Fixed-point Iteration design pattern.
- **Problem:** This pattern is applicable when the problem can be solved stepwise, and a single answer or a subset of input model is returned as a solution.
- **Solution:** The solution can be found in three blocks: (1) an initialization step, which helps to do necessary setup to start solving problem or an initial step towards the solution, (2) a check step, which checks the intermediate form of the input for a possible solution, and (3) an advance step, which iterates the intermediate form of the input one more step closer to the solution.
- **Structure:** Figure 7 depicts the flow and blocks of fixed-point iteration design pattern. The structure forms a while not loop structure. After the initialization step, the control flow does not always have to go to fail case. There are variants of this design pattern, which goes to check step either way after the initialization step (*e.g.,* Dijkstra's algorithm in Section 3.2). The initialization and advance steps are instances of composite rules (CRules), which may include more than one rule.

5. RELATED WORK

We have identified two studies in the literature that introduce reusable structures in model transformation.

Agrawal *et al.* [1] used GReAT language to define three model transformation design patterns. This is the first structured design pattern study in the model transformation area. Each design pattern has *motivation*, *applicability*, *structure*, *known uses*, *limitation*, and *benefits* fields. They introduced *the leaf collector*, which has a visitor pattern [3] like structure and aims to collect or process all leaf nodes in a hierarchy, *the transitive closure*, which can be used to compute the transitive closure of a graph, and *the proxy generator idiom*, which can be used in distributed systems where remote interactions to the system need to be abstracted and optimized.

Iacob *et al.* [4] used QVT Relations language to define five model transformation design patterns. Their design pattern structure has *name*, *goal*, *motivation*, *specification*, *example*, and *applicability* fields. They introduced *the mapping pattern*, which establishes one-to-one relations between elements from the source model and elements from the target model and can be used to translate a model from one syntax to another, *the refinement pattern*, which obtains a more detailed target model by refining an edge or a node to multiple edges or nodes, *the node abstraction pattern*, which abstracts information from source nodes while keeping their relations and can be used to remove elements from models that hold certain criteria, *the duality pattern*, which generates a semantic dual of an instance model, and *the flattening pattern*, which removes the hierarchy from the source model.

These studies are excellent resources, but need to be extended and improved, including some points that need to be analyzed more in future work. First, they are not analyzed in terms of quality. Authors introduce design patterns as the core of their studies, but often do not mention how they affect quality in model transformation problems. Secondly, authors often do not provide a generic way of representing design patterns in terms of a design pattern formalism.

Additionally, we have identified some studies that introduce metrics for ATL [5] and QVT [7] model transformation languages. Amstel and Brand [10] proposed metrics for ATL such as the number of rules with local variables, the number of transformation rules. Kapova *et al.* [6] proposed a set of metrics for declarative transformation languages such as QVT relations language; for example the number of lines of code, the level of inheritance.

6. CONCLUSION & FUTURE WORK

The design of model transformation rules and scheduling suffers from a lack of re-usability. We believe that model transformation design patterns may solve re-usability issues. For this reason, we have focused on one problem and solved the problem in two different ways. Then, we applied the improved solution to solve two other problems with similar structure. Finally, we generalized the solution, which results significantly more efficient metrics, and we have come up with a model transformation design pattern.

We have identified two main points in existing model transformation design patterns that we will analyze for further studies. We show a preliminary measurement with some metrics that affect efficiency. We plan to find more metrics for this purpose. In object-oriented design patterns, the community has agreed to provide design pattern solutions in UML class diagrams [3]. However, there are few studies on this topic and there is no common language or standard for model transformation design patterns. In the design pattern we identified in Section 4, we used MoTif language structures to describe our design pattern. We also plan to generalize the language to a model transformation pattern formalism in order to represent design patterns independently from any rule-based model transformation language.

7. REFERENCES

[1] A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan, and G. Karsai. Reusable Idioms and Patterns in Graph Transformation Languages. In *International Workshop on Graph-Based Tools*, volume 127 of *ENTCS*, pages 181–192, Rome, March 2005. Elsevier.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, Nov. 1994.

[4] M.-E. Iacob, M. W. A. Steen, and L. Heerink. Reusable Model Transformation Patterns. In *Proceedings of the Enterprise Distributed Object Computing Conference Workshops*, pages 1–10, Munich, September 2008. IEEE Computer Society.

[5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[6] L. Kapová, T. Goldschmidt, S. Becker, and J. Henss. Evaluating maintainability with code metrics for model-to-model transformations. In *Proceedings of the 6th international conference on Quality of Software Architectures: research into Practice - Reality and Gaps*, QoSA'10, pages 151–166, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Object Management Group. *Meta Object Facility 2.0 Query/View/Transformation Specification*, jan 2011.

[8] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[9] E. Syriani and H. Vangheluwe. A Modular Timed Model Transformation Language. *Journal on Software and Systems Modeling*, 11:1–28, June 2011.

[10] M. van Amstel and M. van den Brand. Using metrics for assessing the quality of atl model transformations. In *Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011)*, volume 742, pages 20–34, 2011.