The Power of Graph Transformation -
Implementing a Shadow Casting Algorithm

Péter Féher and László Lengyel
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
Email: {feher.peter, lengyel}@aut.bme.hu

Abstract—Nowadays model transformation is used frequently in software development. Therefore, its effectiveness and usability have a big influence on the whole development phase. The attributes of the model transformation are mainly determined by the applied model transformation language. The goal is to develop an expressive, versatile but still effectively processable model transformation language. Using a case study, in this paper the potential of the graph rewriting-based model transformation is presented with respect to some frequent transformation language related problems, e.g. the information sharing between transformation rules. Furthermore, the authors also introduce approaches that make the model transformation languages expressive and summarize the aspects that an efficient model transformation language should address.

Keywords—Model transformation; Model transformation language; Graph transformation

I. INTRODUCTION

Recently model transformation has a bigger role in the software development process. It is applied in a wide spectrum of research field, e.g. Computer Vision [1] or Pattern Recognition [2] [3]. Model transformation is also essential part of the popular Model-Driven Development [4] [5].

Since model transformation is frequently used, it is in the center of model-driven engineering research [6]. There are research results concentrating on how to express model transformations and build appropriate tool support for them. Other research groups are concerned with the development phase of the model transformations, how to specify the different rules and how to implement them.

There are different types of model processing approaches. The traversal-based and direct manipulation approaches provide mechanisms to visit internal representation of the model. They facilitate optimizing, model generation or writing text into streams. These approaches are usually realized by imperative programming languages. The template-based approaches are mainly applied for model-to-code generation. The template contains the mechanism to provide the target text from the source. This can be facilitated either with imperative code or with declarative queries (e.g. OCL [7]). There are also relational approaches, which declaratively map source into target models. This is often achieved by the use of specific constraints. A popular relation approach is the Query/Views/Transformation language (QVT) [8]. The graph rewriting-based approach represents another processing category. In this case, models are represented as typed, attributed, labeled graphs and different graph transformation techniques are used to transform the models. Example approaches are AGG (Attributed Graph Grammar System) [9], AToM³ (A Tool for Multi-formalism and Meta-Modeling) [10], GReAT (Graph Rewriting and Transformation) [11] and the Visual Modeling and Transformation System (VMTS) [12]. Using a structure-driven approach, the transformation is performed in two phases: first a hierarchical structure of the target model is created and next the attributes and references are set. Finally, there are the hybrid approaches which combine some of the previous techniques.

The graph rewriting-based model transformations have the advantage that they are formal and can be applied in many different fields. This wide applicability is well underpinned in this paper. With the help of an illustrative but algorithmically hard case study, the strength of the graph rewriting-based model transformation approaches is proved. The current algorithm is from the field of game development and its goal is to define which cells are visible and which are hidden in a labyrinth from a specific point of view.

In the next section, a solution for this problem is presented in detail with the use of model transformation. The solution is implemented in the VMTS, which is the used model transformation framework. In Section III further related transformations are presented. Finally concluding remarks are elaborated on.

II. THE CASE STUDY

A. The Problem to Solve

In the 1980’s and 1990’s the “rouge-like” games [13] were very popular. Since the user interface was not as sophisticated as today, the games followed a simpler approach: the player got a top-down view of a tile-based world. The original game was called “Rouge” hence the name of the genre. The storyline was light: the player had to go down all levels of the dungeon, find the Amulet of Yendor and get back to the top.

It has a simple approach to lighting the dungeon: everything was visible in the room regardless of whether it was behind an obstacle or not. Later, the rouge-like games had more sophisticated algorithms to determine what was visible. These algorithms took into account the obstacles, the light

...
intensity and so on. In this case study a simpler algorithm is implemented to determine the cells to be shown. It is named as Shadow Casting algorithm [14]. It assumes that all cells in the circle are lit and determines which cells should be in shadow. The basic idea of the algorithm is the following:

1) The table is divided to eight octants. The algorithm gives a solution for the first octant (which is determined by the x-axis and a vector with the direction of 45°). The other octants will be transformed to this octant.
2) The player is transformed to the origin.
3) The algorithm is going to column-by-column from left to right. Within each column the cells are scanned from top to bottom.
4) The algorithm assigns a column portion to the first column. The column portion consists of two directional vectors: an upper vector and a bottom vector. This pair of vectors represents the cells that are not in shadow.
5) At each step, the cells within this region are marked as visible. Furthermore, when an obstacle is found, the column portion is modified.
6) After each column the column portions are assigned to the next column, until the algorithm reached the edge of the circle or there are no columns left.
7) The player and the cells are transformed back to their original place.

Fig. 1 presents an example instance model of the language before and after the transformation with a simple plug-in. The cells with black color are the cells in shadow, the orange brick cells are the obstacles and the smiley face represents the player. The cells with a spider or a blue diamond image represent the spider and diamond objects.

Fig. 1. Example RougeGame model

More information can be found about this algorithm and its implementation in C# in [14] where Eric Lippert examines the algorithm in a series of blog posts. In the next subsections the implementation of the Shadow Casting algorithm is presented with model transformations. First, a domain-specific language is defined by means of a metamodel. Once this metamodel is complete, the transformation rules and the control flow can be specified, which will be introduced in Sec. II-C.

B. The Domain-Specific Language

To be able to create self-evident, expressive transformation rules, a domain specific language (DSL) has to be defined. This DSL is going to specify the domain specific notations and the user-friendly visualization. Since the Visual Modeling and Transformation System (VMTS) is used, a metamodel is created to define this DSL. The metamodel is depicted in Fig. 2.

This language defines the structure of the instance models. The main object is the Table type. This represents the whole world. Every Table type has a relation to a Parameters type. The Parameters type has Radius, PlayerPosX, PlayerPosY and ActualOct attributes. The Radius is responsible for the radius of the circle which represents the actual eyeshot. In this context, a cell is in shadow, if its distance from the Player is greater than the Radius. The Player related properties will be used during the coordinate transformation while in the ActualOct attribute the number of the actually processed octant is stored. Each Table contains Column types. The Column has only a PositionX attribute which stands for its place on the x-axis.

ColumnPortion types can be assigned to the Columns. As it was mentioned before, the ColumnPortion consists of two direction vectors and represents the cells that are not in shadow. It has an attribute named ColumnX which helps to specify which Column the ColumnPortion is related to. The two
direction vectors are the UpperVector and the BottomVector. Both derived from the DirectionVector type which has the SlopeX and SlopeY attributes. Since the vectors pass through the origin, these two properties clearly define the lines.

The Column type can contain Field types. In this case study five different types are derived from the Field: Wall, Floor, Diamond, Spider and Player. Each of these has the PositionX and PositionY attributes to determine its position. Furthermore, they have the Visible attribute, which is enum typed with the values of “Visible”, “Hidden” and “Shadow”, to define their visibility and the IsProcessed attribute which will be used during the processing.

There is also a HelperTable type. The actual processing will take place in the Columns of this table. This is necessary because the ColumnPortion type is assigned only to Columns. This model has some redundant information but they are created for the better performance.

There were other modeling options like using rows, or modeling without Columns but the goal was to stick to the original algorithm as much as possible.

So the model in Fig. 1 has a Table type object which contains nine Columns. Each of these Columns has eleven Fields. The Radius of the eyeshot is four and the Player is in the cell on the 1;1 position.

C. A Model Transformation-Based Solution

After the domain specific language is defined, the transformation rules can be constructed. The example presented in this section introduces a solution that determines which cells are visible and which cells are in shadow from a given player position.

For the better understanding the final control flow is presented first, which is shown in Fig. 3. This model defines how the transformation rules follow each others.

The control flow contains one start state and one or more end state objects. The applicable rules are defined in the rule containers. The rule containers determine which transformation rule has to be applied at the given control flow state. This means that exactly one rule belongs to each rule container. In the rule containers there is also an option to use the global storage to set some elements of the rule before the transformation and/or set the variable in the global storage with an element from the rule after the transformation. The number of the rule application can also be defined here. By default, the VMTS tries to find just one match for the left-hand side (LHS) of the transformation rule. However, if the IsExhaustive attribute of the rule container is set to true then the rule will be applied repeatedly as long as its LHS pattern can be found.

The edges are used to determine the sequence of the rule containers. The control flow follows an edge in order of the result of the rule application. In VMTS, the edge to be followed in case of successful rule application is depicted with a solid gray flow edge, and in case of failed rule application with a dashed gray flow edge. Solid black flow edges represent the edges that can be followed in both cases.

In VMTS, the left-hand side and the right-hand side (RHS) of the transformation rule are represented together. This way the transformation itself might be more expressive. In order to distinguish the LHS from the RHS, the VMTS uses different colors. The elements represented with blue color are created by the transformation rule. This means if the LHS and the RHS would be depicted in two separated graph, these elements
Fig. 3. The control flow model of transformation SetCellVisibility

would be only part of the RHS graph. Similarly, the red color indicates that the given element will be deleted by the transformation rule. The yellow color is used when an edge between two elements will be replaced. In this case the type and the attributes of the edge will not change. The gray background indicates that the element will be modified. With the help of these colors the transformation process is easily understandable. There is always an option to use imperative constraints to each element, but it is not depicted separately.

First the Parameters object has to be set. This rule only sets the ActualOct attribute to one. The next step is to initialize the Field objects. This means that the Visible attribute is set to “Hidden” for all cells. In the SetPlayerRule rule the position of the Player object is saved in the appropriate property of the Parameters object. The final two steps of the initial phase are related to the HelperTable. First it is created then some Column objects are added as well. The number of the created Columns is equal to the size of the eyeshot that is to the Radius attribute.

After this, the Field objects are created in the Column of the HelperTable. These Fields have to be the exact copy of the appropriate Fields in the original Column after the coordinate-transformation. The ActualOct attribute of the Parameters object helps to decide which transformation should be used. For the sake of simplicity, Fig. 3 depicts only the transformation of the Wall and Floor objects. The TransformFloorRule is shown in Fig. 4.

In VMTS there is an option to add imperative constraint and/or imperative code to the transformation rule. In this case it is determined in the imperative constraint which cells of the original Table have to be transformed. This is done by the help of the ActualOct attribute of the Parameters. As it was mentioned earlier, the ActualOct attribute also used to the transformation as well. Finally, the PositionX and PositionY attributes of the newly created Field are set in the imperative code. Here the position of the Player object is also taken into account, it is transformed to the “origin”.

After the Columns are filled up with the transformed Fields, it is time to determine which one is “Visible” and which one is “Hidden”. For this, there is a need for an initial ColumnPortion. This is created in the Create1stCPRule. The ColumnPortion is assigned to the first Column. The attributes of the two DirectionVector are also set: The value of the SlopeX and SlopeY attributes will be 1 and 1 for the Up- perVector and 1 and 0 for the BottomVector, respectively.

Now everything is set to determine the visibility of the cells. This is done in the SetVisibilityRule. Earlier the ColumnPortion object was defined as it contains the cells that are not in shadow. This means that every cell in the Column which is between the UpperVector and BottomVector of a ColumnPortion is visible. So the LHS of the rule is looking for Fields in the column which has ColumnPortion objects and meets the criteria above.

The next six rules are the soul of the algorithm. They are looking for obstacle objects which modify the ColumnPortion objects. In this case study there is only one obstacle object, the Wall. Since the cells are scanned from top to bottom, the uppermost Wall has to be found. This is done by the GetWallRule and GetUpperWallRule rules. First a Wall object is needed which is not processed yet. Then the rules are repeatedly looking for another Wall in the same Column which has bigger PositionY value.

When the uppermost Wall is found, the corresponding ColumnPortion has to be modified. There are four different
cases:

1) The next upper cell is not part of the ColumnPortion, i.e. “Hidden”. In this case only the attributes of the UpperVector have to be modified. This is done by the UpdateCPSimpleRule. The rule is shown in Fig. 5.

2) The next upper cell is a Wall. This time the UpperVector has to be modified, like in the case above. The UpperCPDoubleRule is responsible for this.

3) The next upper cell is a visible, not obstacle cell. This time the transformation needs to create another ColumnPortion as well. The old ColumnPortion will get a new UpperVector while the BottomVector remain the same. The newly created one will get the UpperVector from the old one and the BottomVector will be set in accordance with the Wall. The UpdateCPRule deals with this situation.

4) There is no upper next cell to the Wall. In this case the transformation has to do the same as in the first two cases.

The above order is important since the rules were created according to them. Once a rule is found that has a match, the transformation modifies the ColumnPortion, sets the Wall object as processed and goes back to the GetWallRule to find another Wall. It stays in this loop until there is no unprocessed Wall left.

During modifying the ColumnPortion, there is a possibility that the gradient of the UpperVector will be less than the gradient of the BottomVector. The transformation does not have to deal with these ColumnPortions so it deletes them in the DeleteBadCPRule.

In the next step, the transformation has to check that how far it is from the origin. If the number of the actual Column is less than the Radius attribute then the ColumnPortions is moved to the next column by the MoveCPRule. In case the checked column is on the edge of the eyeshot, then the visibility process is finished. This means that the cells from the HelperTable have to be transformed back to the original Table. The TransformVisibleRule set the Visible attribute of the original Field in case the cell in the HelperTable is “Visible”. The rule is based on the same transformation that was used the other way. To improve the performance, during the TransformFloorRule and the TransformWallRule the transformed Fields are put into a list in the global storage. Now, when these cells are transformed back, the transformation can use this list to reduce the matching time, thus improve the performance. In VMTS, when the global storage is used to set cells in the LHS, the Iterative checkbox can be checked in the rule container. This way the engine tries to match the LHS for all elements in the list.

Because there is no need for the ColumnPortions, Fields and Columns in the HelperTable, the transformation deletes them.

In the UpdateOctRule the value of the ActualOct attribute is examined. If it is less than eight, then the transformation increases it and goes back to the CreateColumnsRule to start the loop again. In case it finished with the last octant, the HelperTable is deleted and the transformation terminates.

III. FURTHER TRANSFORMATIONS

In the previous section an approach was presented which solves the problem of the “cell visibility”. In order to form a whole model, in the following some simpler transformations are briefly introduced to complement the previous one.

A. Moving the Player

The player will move throughout the table, so there is a need for a transformation that achieves this. When we are talking about moving an object we mean that the types of the affected cells are going to change. The cell the object leaves will be a
Floor typed cell, while the cell where it steps will be a Player typed one. The rule is depicted in Fig. 6.

In this example, to decide where the player is going to step, a random generator is used. This is achieved in the imperative constraint related to the rule. Depending on the randomly generated number the transformation examines the attributes of the next cell. Therefore, the control flow of this transformation is fairly simple, since it just repeatedly applies the rule until the randomly generated number and the next cell match.

B. Moving the Spiders

The other object type that can be moved is the Spider. There is just one difference between the two transformation rules, namely the type of the cell the rules try to match. In this case the rule is looking for Spider objects. The direction of the move is decided the same way as in the previous rule, with randomly generated numbers. There is possibility to apply artificial intelligence to decide the direction of the Spiders but for the sake of simplicity this is not implemented in this case study.

C. Create New Objects

There might be a valid claim to create new objects. For example we want to increase the number of Spiders in regard to the elapsed time or to create a new Diamond after the player just found one. To accomplish that, the CreateObjectsRule is implemented. This rule creates a new Diamond object. Similarly, the CreateSpiderRule is responsible for creating new Spiders. These simple rules change the type of the selected cells.

IV. CONCLUSION

In this paper the authors demonstrated the usability of the model transformations throughout a case-study realized in the Visual Modeling and Transformation System (VMTS). Since there is a strong tool support, it can be used successfully and effectively in many different situations.

During the implementation different language elements were used. The information sharing between the transformation rules was realized with the help of the global storage and to increase the matching performance initial matches were set. In the control flow a sequential order of the transformation rules were developed with conditional branching and iterations. Considering this, it takes the following two aspects to create effective model transformations:

1) The usage of an expressive model transformation language to define convenient, self-evident transformation rules.
2) The effective and correct application of the transformation definition for the input models.

Currently the extended version of VMTS transformation language is being developed. In order to support the previous requirements, this language has the following capabilities:

• Support for language elements, e.g. multiplicity and abstraction to increase the abstraction level of the transformation rules.
• Solution for defining constructions both declaratively and imperatively.
• Support for pivot nodes in order to increase the efficiency of the matching process.
• Implementation of a type safe local and global storage for the information sharing.
• Control flow capabilities: sequential order of rules, conditional branching, iteration and recursion.
• Support for developing hierarchic control flows and control flows which are capable of parallel execution.
• Solution for parameter passing.
• Support for defining patterns and applying pre-defined, optimized patterns.
• Possibility for implementing generic structures for the purpose of increasing the abstraction level.
• Solution for debugging features in order to effectively support the development of the models.

The language addresses these aspects, supports the effective model transformation processing, and makes it possible to define models in an expressive, intuitive way.

ACKNOWLEDGMENT

This work is connected to the scientific program of the “Development of quality-oriented and cooperative R+D+I strategy and functional model at BUTE” project. This project is supported by the New Hungary Development Plan (Project ID: TAMOP-4.2.1/B-09/1/KMR-2010-0002).

REFERENCES