Flexible Visualization of Automatic Simulation based on Structured Graph Transformation

Enrico Biermann, Claudia Ermel, Jonas Hurrelmann
Inst. für Softwaretechnik und Theoretische Infomatik
Technische Universität Berlin
Franklinstr. 28–29,
D - 10587 Berlin, Germany
{enrico,lieske,jhurrel}@cs.tu-berlin.de

Karsten Ehrig
Federal Institute for Materials
Research and Testing (BAM), Berlin
Unter den Eichen 87
D - 12205 Berlin, Germany
karsten.ehrig@bam.de

Abstract

Visual modeling languages for discrete behavior modeling allow the modeler to describe how systems develop over time during system runs. Models of these languages are the basis for simulation with the purpose to validate the model with respect to its requirements. Graph transformation systems have shown to be suitable for the definition of various kinds of visual modeling languages. They define a model's operational semantics as the set of all transformations of a model which are specified by graph transformation rules. For automatic simulation, rules have to be structured to control their application order. During simulation, the state changes after each rule application should be visualized in the concrete syntax of the modeling language.

In this paper, we propose a generic approach to specify simulation environments based on a model's concrete syntax definition and suitable rule structuring techniques. We implement our approach using TIGER*, a tool for defining visual languages based on graph transformation, and generate the specified simulation environment as plug-in for* ECLIPSE*. We demonstrate our approach by a case study for automatic simulation of Rubik's Clock, a mechanical two-sided puzzle of clocks controlled by rotating wheels.*

1. Introduction

Simulation is usually defined as "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system " [16].

For visual behavior modeling languages such as statecharts or Petri nets, usually simulation means to explore step by step (hand-triggered or automatically) the state space of a model. Simulation thus is based on a step semantics of the modeling language.

Graph transformation systems have shown to be suitable for the definition of various kinds of visual modeling languages. Step semantics is defined for graph transformation systems in a formal way as the set of all transformations of a model, where the possible transformations are specified by the graph transformation rules. We call such a graph transformation system for simulation a *simulation specification* [9]. In order to allow *automatic* simulation, rules in a simulation specification may be structured to control their application order. This can be done either by using transformation units [13], rule layers [1] or rule priorities [3].

Approaches exist which combine model manipulation based on graph transformation rules with model visualization in a certain concrete syntax, like e.g. (TIGER [19], AToM^3 [4], and DIAGEN [15]). These tools generate visual editors which also allow for a rudimentary simulation, but for concrete syntax manipulation they mainly rely on programmed layouters. Furthermore, rule application control also is realized by changing the Java code of the generated editors by hand.

In this paper, we propose an approach for simulation specification which, on the one hand, supports the manipulation of concrete syntax elements directly in the simulation rules, and, on the other hand, allows for rule control structure definition, where rule control expressions (e.g. rule sequences rule application as long as possible or priorities) are integrated with graph constraints to control the order and conditions of rule applications for automatic simulation runs. This facilitates the generation of simulation environments, i.e. model-based prototypes which help to validate model behavior using a convenient domain-specific visualization of the model states.

As running example, we specify the automatic solution of Rubik's Clock, a mechanical puzzle of clocks controlled by rotating wheels. Rubik's Clock [14] is a two-sided puzzle, each side presenting nine clocks to the puzzler (see Fig. 1). There are four wheels, one at each corner of the puzzle, each allowing the corresponding corner clock to be rotated directly. (The corner clocks, unlike the other clocks, rotate on both sides of the puzzle simultaneously and can never be operated independently. Thus the puzzle contains only 14 independent clocks.)

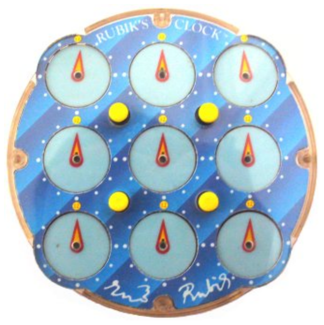


Figure 1. Front Face of Rubik's Clock

There are also four buttons which span both sides of the puzzle; each button arranged such that if it is ”up” on one side it is ”down” on the other. The state of each button (up or down) determines whether the adjacent corner clock is mechanically connected to the three other adjacent clocks on the front side or on the back side: thus the configuration of the buttons determines which sets of clocks can be turned simultaneously by rotating a suitable wheel. The aim of the puzzle is to set all nine clocks to 12 o'clock (straight up) on both sides of the puzzle simultaneously.

The paper is structured as follows: In Section 2 we develop simulation rules for Rubik's clock and discuss control structures. Section 3 sketches the implementation of the presented approach in TIGER. In Section 4, we compare our approach to related work, and in Section 5, we conclude the paper with ideas for future work.

2. Simulation by Graph Transformation

Graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation [7, 8]. Especially for the application of graph transformation techniques to visual language (VL) modeling, *typed attributed graph transformation systems* [7] have proven to be an adequate formalism.

2.1 Visual Language Modeling

A VL is modeled by a type graph defining the underlying visual alphabet, i.e. the symbols and relations which are available. Sentences or diagrams of the VL are given by graphs typed over (i.e. conforming to) the type graph. Such a VL type graph corresponds closely to a meta model.

Example 2.1 (*Rubik's Clock VL*)
Fig. 2 shows the abstract syntax type graph of the Rubik's Clock model. A *Clock* has an attribute *wheel* which indicates whether this clock has a wheel (i.e. it is a corner clock). The time on the front side of the clock is indicated by *time_front* and the time on the back side of the clock is indicated by *time_back*. Please note that *time_front* = *time_back* for all wheel clocks, whereas the front and back times are independent for all non-wheel clocks. For simplicity we have chosen *integer* as type for the time type. Therefore, all clock times are calculated modulo 12.

The attributes *step_front* and *step_back* are used for simulation to store the difference between the clock time before and after a simulation step of the clock. E.g., if the front clock shows one o'clock and is set to three o'clock, then the value of *step_front* becomes 2. A *Button* has the boolean attribute *up* which indicates whether the *Button* is up ($up = true$) or down ($up = false$). *Clocks* and *Buttons* are connected via the link type *connection*.

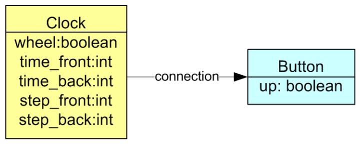


Figure 2. Abstract Syntax Type Graph

Fig. 3 shows the initial state of Rubik's Clock according to Fig. 1 as graph in abstract syntax typed over the type graph in Fig. 2. All time and step attributes are set to zero, meaning that no wheel has been turned and that all clocks are showing the time twelve o'clock (hands pointing upwards). Initially, all buttons are up.

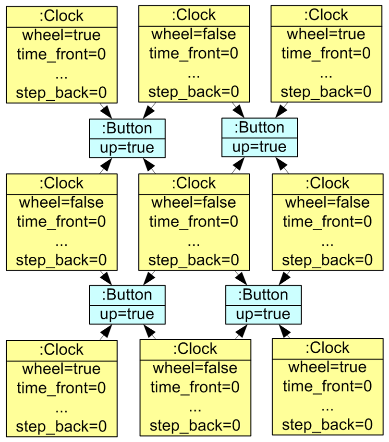


Figure 3. Initial State in Abstract Syntax

2.2 Rule-based Simulation

On the basis of a type graph defining a behavioral visual language, and graphs typed over this type graph representing different states of a model, step-wise simulation is now described by graph transformation between these states. The main idea of graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. The core of a graph transformation rule $(LHS \xrightarrow{p} RHS)$ is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective (partial) graph morphism $p : LHS \rightarrow RHS$. Applying the rule $(LHS \xrightarrow{p} RHS)$ means to find a match of LHS in the source graph and to replace this matched part in the source graph by the corresponding RHS, thus transforming the source graph into the target graph of the graph transformation.

Intuitively, the application of rule r to graph G via a match m from LHS to G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing condition* is satisfied, i.e. the deletion step must not leave *dangling edges*, and for two objects which are identified by the match, the rule must not preserve one of them and delete the other one.

Definition 2.2 (*Graph Transformation*)
Let $(LHS \xrightarrow{r} RHS)$ be a typed graph transformation rule and G a typed graph with a typed graph morphism $LHS \xrightarrow{m} G$, called match. A *graph transformation step* $G \overset{r,m}{\Longrightarrow} H$ from G to a typed graph H via rule p, match m, and co-match m^* is shown in the diagram to the right. The rule r may be extended by a set of *negative application conditions* (NACs) [12, 7]. A match $LHS \xrightarrow{m} G$ satisfies a NAC with the injective NAC morphism $n : LHS \rightarrow NAC$, if there is no injective graph morphism $NAC \xrightarrow{q} G$.

$$\begin{tikzcd}
NAC \arrow[drr, "q"', "/" marking] & & LHS \arrow[ll, "n"'] \arrow[r, "r"] \arrow[d, "m"'] & RHS \arrow[d, "m^*"] \\
& & G \arrow[r] & H
\end{tikzcd}$$

A sequence $G_0 \Rightarrow G_1 \Rightarrow .. \Rightarrow G_n$ of graph transformation steps is called *graph transformation*, denoted as $G_0 \overset{*}{\Rightarrow} G_n$.

A rule may be extended by input parameters, i.e. variables used to compute new attribute values for nodes in the right-hand side. When the rule is applied, the input parameters have to be bound to concrete values.

Example 2.3 (*Simulation Rules for Rubik's Clock*)
Fig. 4 shows some simulation rules for Rubik's Clock in abstract syntax. Rule *turn* is the initial rule for simulation. A value for input parameter *time* is required from the user, stating the time to which the wheel clock is turned in this simulation step. In the RHS of the rule, values for *time_front* and *time_back* are set to the *time* parameter. The number of steps the wheel clock has been turned is computed by adding the *time* value to the previous front or back time numbers. The step number is stored in *step_front* and *step_back* to be used to turn adjacent clocks accordingly.

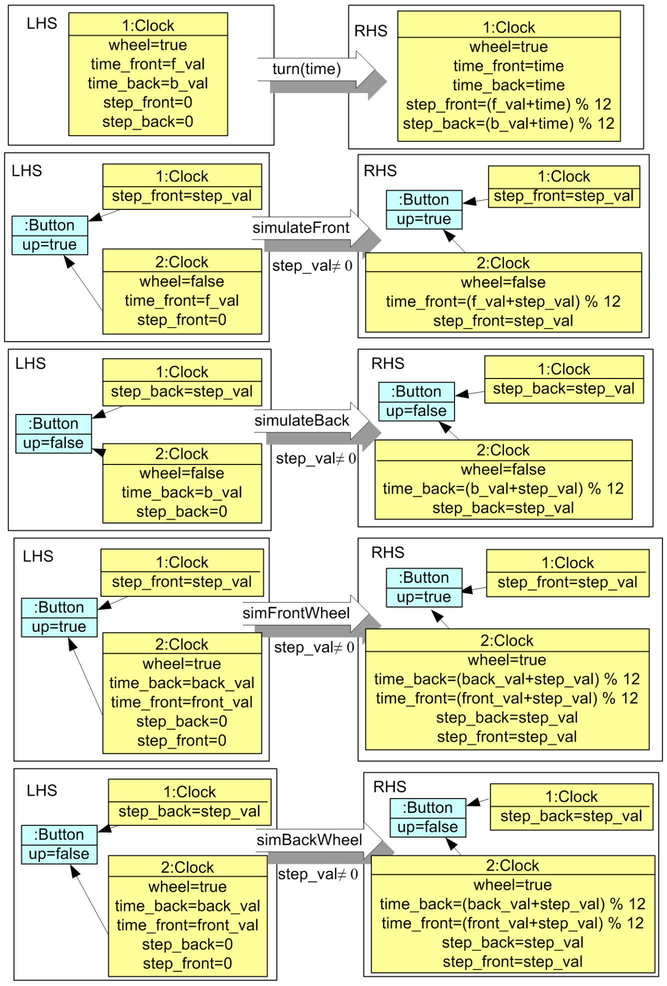


Figure 4. Simulation Rules for Rubik's Clock

Rule *simulateFront* in Fig. 4 increments the time of a front clock if it's not a wheel clock and the button is up. The value of *time_front* of clock No. 2 is set to *(front_val + step_val) modulo 12*, where *front_val* is the previous front time, and *step_val* is the number of steps that clock No. 1 has been turned before. The value of *step_front* of clock No. 2 is set to *step_val*, i.e. the number of steps the clock has been turned. An application condition forbids the rule application if the value of *step_front* is different from zero. Rule *simulateBack* is defined analogously to rule *simulateFront* and sets the back time of a non-wheel-clock if the connected button is in "down" position. Similar rules *simFrontWheel* and *simBackWheel* exist to increment both the front and the back clock time of a connected wheel clock in parallel when the button is up or down. In the first case, the front clock step is copied from the neighbor clock, in the second case the back clock step. Finally, two rules *resetStepFront* and *resetStepBack* (not depicted) reset all *step_front* resp. *step_back* variables to zero. These variables indicate the number of forward ($step > 0$) or backward ($step < 0$) moves during one simulation step.

2.3 Defining Control Structures for Rules

One hand simulation step (turning the wheel of a clock and adjusting all resulting time values for all other clocks) consists of applying once the initial rule (*turn(time)*) and afterwards applying the simulate rules (*simulateFront, simulateBack,..*) as long as they are applicable. At last, the step values of all clocks are reset by applying the reset rules:

- *turn(time);* (apply only once)
- *simulateFront, simulateBack, simFrontWheel, simBackWheel;* (apply as long as possible)
- *resetStepFront, resetStepBack.* (apply as long as possible)

For solving clocks, i.e. applying a set of simulation rules to turn one clock to 12 without changing the times of all other clocks, a more high level application structure (e.g. transaction systems) is needed. Fig. 5 shows the control flow for solving a complete Rubik's Clock. We distinguish between three types of clock:

1. *Wheel Clocks* are corner clocks with the attribute $wheel = true$;
2. *Border Clocks* are clocks on the border of Rubik's Clock which are not *Wheel Clocks*;
3. *Center Clocks* are clocks inside Rubik's Clock.

Solving a clock means to apply simulation steps in such a way that in the result the selected clock is turned to 12 and the times of all other clocks remain unchanged. Both sides of the clocks are treated separately, i.e., if the front side of a *Border Clock* is solved the back side remains unchanged and has to be solved in a separate step. One exception exists for *Wheel Clocks*; here front and back side is connected via the wheel and so solving has to be applied only once.

3. Implementation of Rubik's Clock Simulation in TIGER

TIGER [2] is a visual environment to design visual language (VL) specifications based on meta models, graph

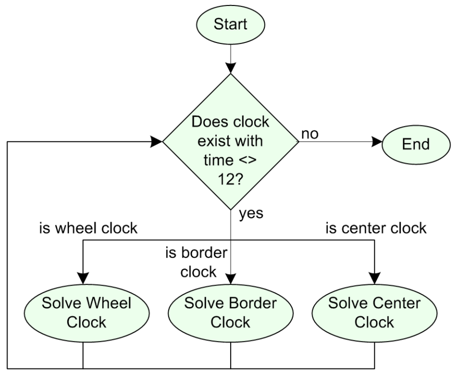


Figure 5. Control Flow for Solving Clocks

grammars and layout definitions. A VL specification serves as basis to generate the Java source code of a visual editor for VL diagrams as ECLIPSE plug-in based on the Graphical Editor Framework GEF [5].

3.1 Editor Generation by TIGER

A VL specification (abstract syntax type graph, concrete syntax figures and editor commands) is defined visually by the language modeler using the TIGER Designer [10]. Editor commands are modeled as graph rules. The application of such graph rules to the underlying syntax graph of a diagram is performed by the graph transformation engine AGG [18]. In TIGER, the abstract syntax graphs manipulated by AGG are extended by means for concrete syntax definition. TIGER distinguishes *node symbols types* and *edge symbol types* in a VL specification. The concrete syntax defined for node symbol types (such as shape, fill color, and size of a figure for an abstract model element) are translated to features and graphical constraints of *GEF figures*. Analogously, the concrete syntax for edge symbol types (such as color, thickness or arrowheads of an edge figure) are translated to properties of corresponding *GEF connections*. Thus, the generated editors make use of a variety of GEF's predefined editor functionalities and appear in a timely fashion, conforming to the ECLIPSE standard for graphical tool environments.

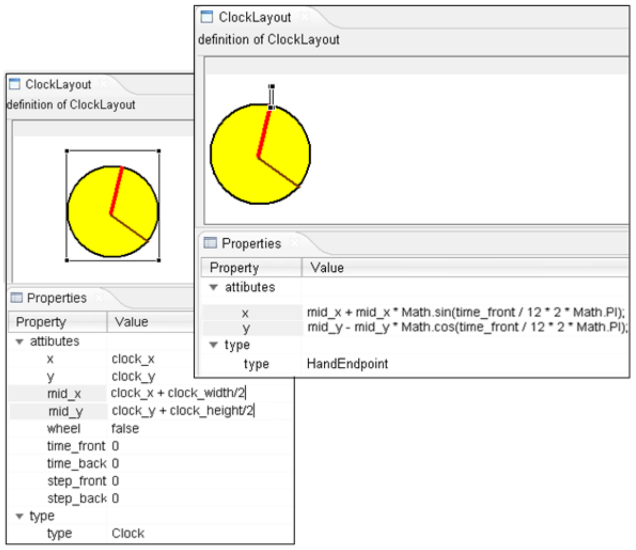
3.2 Extending TIGER for Flexible Visualization

Up to now, the only concrete syntax attribute, which can be changed or defined in graph rules, is the position of a node-type shape (x and y coordinates). Other properties such as size and color are fixed for all shape figures corresponding to a node symbol type. Moreover, a model element may be visualized by a complex figure which consists of a combination of several simple figures, but then this complex figure is invariable (such as the end state of a state machine which invariably consists of a larger white circle containing a smaller black circle in its center).

For simulation, this rather simple and static one-to-one relation between abstract model elements (e.g. Clock) and their visual representation (e.g. Ellipse) is in general not adequate. What we need for our Rubik's Clock simulation is the possibility to model a clock's hands (showing the front and the back time) and to change the position of these hands within the clock circle.

Hence, we propose in this paper an extension of the TIGER Designer by more flexible means to define and manipulate concrete syntax properties. On the one hand, we will support the definition of more than one graphical figure for an abstract model element in a way that the properties of the graphical figures depend on the current values of available abstract attributes of the model element; on the other hand, the user may also manipulate the properties of concrete figures using the graph rule editor.

According to these extended possibilities, the concrete syntax for editing a clock model element may now be defined conveniently in the property view of the syntax graph layout editor as shown in Fig. 6.



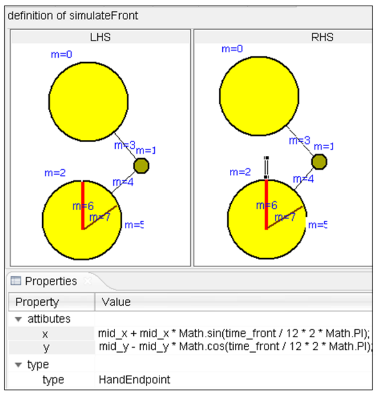
Property	Value
▾ attibutes	
x	clock_x
y	clock_y
mid_x	clock_x + clock_width/2
mid_y	clock_y + clock_height/2
wheel	false
time_front	0
time_back	0
step_front	0
step_back	0
▾ type	
type	Clock

Property	Value
▾ attibutes	
x	mid_x + mid_x * Math.sin(time_front / 12 * 2 * Math.PI);
y	mid_y - mid_y * Math.cos(time_front / 12 * 2 * Math.PI);
▾ type	
type	HandEndpoint

Figure 6. Visualization of a *Clock*

A clock circle (the main figure for the model element *Clock*) is defined together with its front and its back hands. In the left screenshot, the main clock circle figure has been selected, and it can be seen in the property view that here all abstract clock attributes are defined, together with the concrete x and y coordinates of the upper left corner of the clock circle figure. Moreover, the center point (mid_x, mid_y) of this circle are computed. The parameters $clock_x$ and $clock_y$ are the mouse click coordinates denoting the position where the clock is placed when the rule is applied. The definition of the endpoint coordinates (x, y) of a front-time hand is shown in the property view in the right screenshot in Fig. 6. The position of the hand's endpoint depends on the value of the abstract *time_front* attribute.

Alternatively, the position of a clock hand's endpoint may be defined in the corresponding simulation rules for creating clocks and for adjusting their time (see Fig. 7).

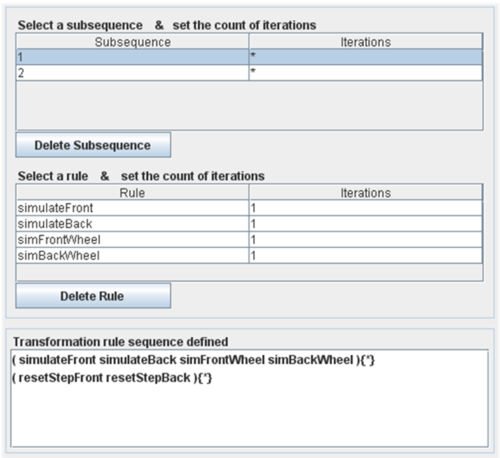


Property	Value
▾ attibutes	
x	mid_x + mid_x * Math.sin(time_front / 12 * 2 * Math.PI);
y	mid_y - mid_y * Math.cos(time_front / 12 * 2 * Math.PI);
▾ type	
type	HandEndpoint

Figure 7. Graph Rule *simulateFront*

3.3 Extending TIGER for Controlled Simulation

For controlling the application of simulation rules as described in Section 2.3, we need a wizard to define units as sequences of rules and to define how often the sequences within a unit should be applied. Fig. 8 shows such a wizard where the unit *simulateWheelTurn* is defined.


Select a subsequence & set the count of iterations

Subsequence	Iterations
1	*
2	*

Select a rule & set the count of iterations

Rule	Iterations
simulateFront	1
simulateBack	1
simFrontWheel	1
simBackWheel	1

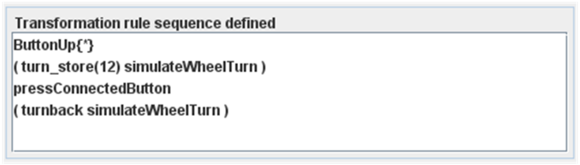
Transformation rule sequence defined

```
( simulateFront simulateBack simFrontWheel simBackWheel ){*}
( resetStepFront resetStepBack ){*}
```

Figure 8. Defining Unit *simulateWheelTurn*

The asterisk "*" after a rule sequence means that all rules in this sequence are applied as long as possible. Note that rule *turn* is not part of this unit, since we want to reuse *simulateWheelTurn* after different variations of the *turn* rule in the more complex units *SolveWheelClock*, *SolveBorderClock* and *SolveMiddleClock*, according to Fig. 5.

As example, Fig. 9 shows the transformation unit solveWheelClock, where the previously defined unit *simulateWheelTurn* is used whenever a wheel clock has been turned. At first, all buttons are put in "up" position by rule *ButtonUp*, which is applied as long as possible.



Transformation rule sequence defined

```
ButtonUp{'}
( turn_store(12) simulateWheelTurn )
pressConnectedButton
( turnback simulateWheelTurn )
```

Figure 9. Unit *solveWheelClock*

In order to store the value of the step variable from the first wheel clock, the *turn* rule is slightly extended here (see rule *turn_store* in Fig. 10) by a value-storing node of type *Store-Step* which is connected to the turned wheel clock. Afterwards, the unit *simulateWheelTurn* is applied to adjust all clocks depending on the steps the wheel clock has been turned and on the position of the buttons. In the next rule *pressConnectedButton*, the new value-node indicates which wheel clock is the active one (see Fig. 10). Here, the button connected to the active wheel clock is pressed. At last, another wheel clock is turned as many steps as the number in the value-node. By a negative application condition, it is ensured that not the first wheel clock is turned back. Again, unit *simulateWheelTurn* adjusts all other clocks after the second wheel clock has been turned.

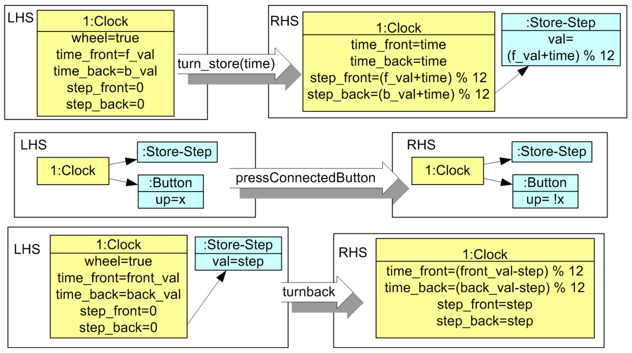


Figure 10. Rules in Unit *solveWheelClock*

3.4 Generated Editor and Simulator

Fig. 11 shows the generated Rubik's Clock editor plug-in in ECLIPSE. In the editor palette, editing operations *createClock*, *createWheelClock* and *createButton* can be selected to insert *Clocks*, *WheelClocks* (stronger border layout) and *Buttons* respectively. Rule *connect* allows to draw a connection between *Clocks* (of arbitrary types) and *Buttons*. After editing is complete, the simulation rule *turn* can be applied to one *WheelClock*. The number of steps to be turned has to be entered into an input dialog window.

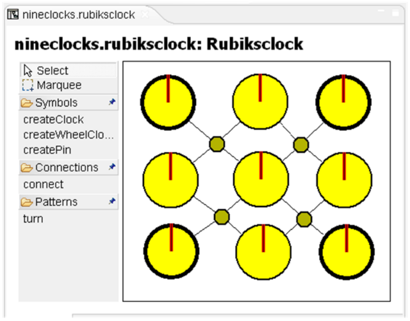


Figure 11. Rubik's Clock Editor Plug-in

For solving it is suitable to define relative position values (x, y) for clocks and pins. Starting with position (1,1) for the clock and pin in the left top corner we end up with position (3,3) for the clock and (2,2) for the pin in the right bottom. Please note that the solving algorithm is suitable for rectangular (m, n) clocks in general with $m, n > 2$. For $m = 2$ or $n = 2$ we have the special cases which may be treated separately, e.g. the clock with $m = n = 2$ consists of 4 *WheelClocks* which have always the same time on front and back.

Fig. 12 shows the solving steps for the *WheelClock* (1,1):

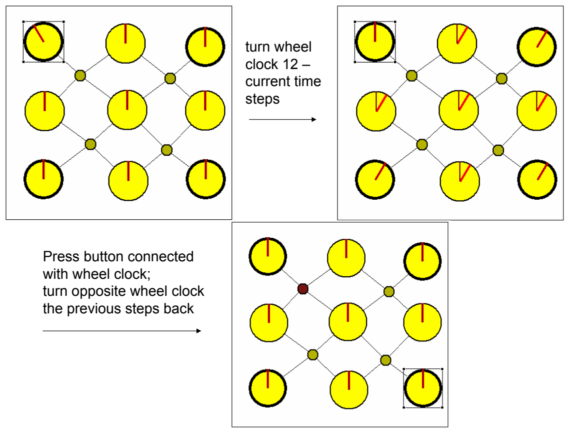


Figure 12. Solving Steps for Wheel Clocks

1. Pull all buttons up;
2. Turn *WheelClock* to $time = 12$;
3. Press button connected with *WheelClock*;
4. Turn *WheelClock* on another corner the previous number of steps back.

Fig. 13 shows the solving steps for the *BorderClock* $(cx, cy) = (2, 1)$:

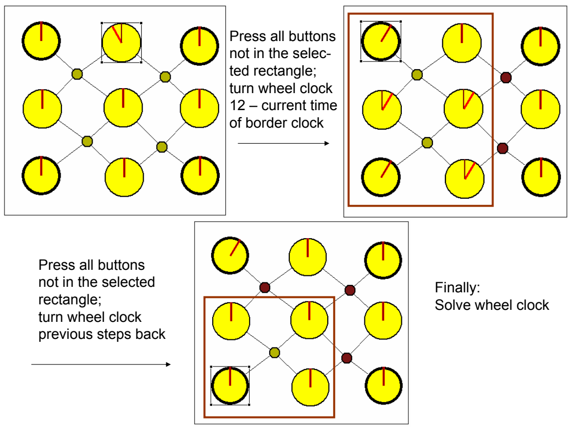

Press all buttons not in the selected rectangle; turn wheel clock 12 – current time of border clock
Press all buttons not in the selected rectangle; turn wheel clock previous steps back
Finally: Solve wheel clock

Figure 13. Solving Steps for Border Clocks

1. Pull buttons (bx, by) with $bx < cx$ up (and push all others down);
2. Turn *WheelClock* (1,1) *(12 - currentTime)* of *BorderClock*;
3. Press button (bx, by) with $bx = cx - 1$ and $by = 1$;
4. Turn *WheelClock* $(1, n)$ the previous steps back;
5. Solve *WheelClock* $(1, 1)$.

Fig. 14 shows the solving steps for the *CenterClock* $(cx, cy) = (2, 2)$:

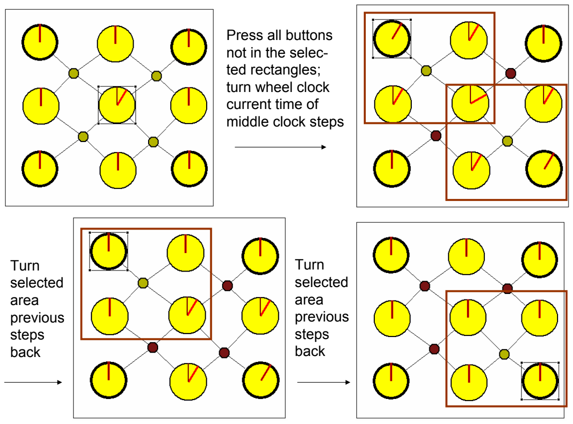

Press all buttons not in the selected rectangles; turn wheel clock current time of middle clock steps
Turn selected area previous steps back
Turn selected area previous steps back

Figure 14. Solving Steps for Center Clocks

1. Pull buttons (bx, by) with $bx < cx, by < cy$ and $bx >= cx, by >= cy$ up (and push all others down) ;
2. Turn *WheelClock* $(1, 1)$ *currentTime* of *CenterClock* steps forward;
3. Pull buttons (bx, by) up with $bx < cx, by < cy$ (and all others down);
4. Turn *WheelClock* $(1, 1)$ previous steps back;
5. Pull buttons (bx, by) with $bx >= cx, by >= cy$up (and push all others down);
6. Turn *WheelClock* (m, n) previous steps back.

Due to symmetry, all other clocks can be solved by rotating Rubik's Clock such that the current clock matches one of the positions above.

3.5 Out-Shaped Clocks

All regular clocks in rectangle format with (m, n) dimension ($m, n \in \mathbb{N}$ and $m, n > 2$) can be solved using the solving algorithm above. However, the editing rules support also creation of more irregular clocks which we call *out-Shaped clocks*. Fig. 15 shows a sample *out-shaped clock* in triangle format. Since the hand-simulation rules shown in Fig. 4 work for these clocks as well, the Rubik's Clock editor plug-in provides a nice and easy-to-use environment to experiment with other irregular forms of clocks to find a more general solving algorithm.

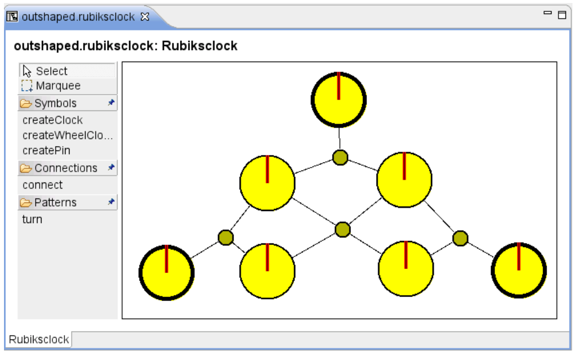


Figure 15. Out-Shaped Clock in Triangle Form

4. Related Work

The Rubik's Clock puzzle was invented and patent-registered in 1989. Since this time several websites have been created presenting solving algorithms which are quite similar to this one presented in this paper (see e.g. `http://www.geocities.com/jaapsch/puzzles/clock.htm`). Moreover, the Rubik's Clock puzzle was also investigated by Sugiyama et al. in [17] dealing with the layout of puzzle graphs. However, none of the known solving algorithms deal with general (m, n) rectangular clocks, nor with irregular clocks like the *out-shaped clock* briefly discussed in Section 3.5.

Existing tools for graph transformation-based simulation (AToM^3 [4], DIAGEN [15]) generate visual editors from a meta model-based VL definition, which is combined statically with a concrete syntax definition for the VL elements. The visual editors also allow for a rudimentary simulation but up to now they do not support a systematic manipulation of concrete syntax attributes, such as position, size or color of shapes in the simulation rules. Furthermore, rule application control (apart from priorities used in AToM^3) is realized by changing the code of the generated environment for editing and simulation by hand.

5. Conclusion and Future Work

The Rubiks's Clock puzzle proved to be a nice case study to analyze flexible visualization and the use of structured control units for automatic simulation. The TIGER user may access abstract and concrete attribute values when designing the visual appearance of model elements and when defining simulation rules. Together with the possibility to choose more than one independent figures for one model element, this results in a high degree of flexibility in visualization. Thus, clock time values can be visualized dynamically without exchanging the figure for each time change.

As future work, a clearer separation of abstract and concrete syntax attributes would be nice, such as two separate property views for the "core" model (the abstract syntax) and the "appearance" (the concrete syntax), as used e.g. in the ECLIPSE Graphical Modeling Framework GMF [11].

Our second extension of TIGER enables the user to define automatic simulation runs in form of control units on rules, such as rule sequences, conditions and counters for rule applications. With this extension, we can specify a general automatic simulation of different kinds of rectangular Rubik's Clocks and smaller simulation steps for out-shaped ones. Defining structured control units for graph transformation rules still lacks the flexibility from usual programming since each rule in a unit operates on the host graph independently of the other rules. Hence, the match of a rule cannot be handed over to the next rule of the sequence, but has to be marked indirectly in the host graph. Analogously, attribute values computed by one rule have to be stored in distinct node attributes, where the next rule can find them. Therefore, our simple hand-simulation rules for Rubik's Clock had to be extended by further attributes in order to store match information for the next rule of the unit. Here, more general concepts like global pattern parameters for units should be considered in future to come up with an even more practicable structuring concept for graph transformations. Furthermore, work is in progress to integrate an activity diagram editor in the TIGER *Designer* component to define control structures visually instead of using the text-based wizards presented in this paper.

References

[1] R. Bardohl, C. Ermel, and I. Weinhold. Graph Transformation-Based Analysis Techniques for Efficient Visual Language Validation. *Proc. Graph Transformation-Based Tools (GraBaTs'02)*, pages 120–130, 2002.

[2] E. Biermann, K. Ehrig, C. Ermel, and G. Taentzer. Generating ECLIPSE Editor Plug-Ins using TIGER. *Proc. Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, LNCS, Springer, to appear 2008.

[3] J. de Lara. Meta-Modelling and Graph Transformation for the Simulation of Systems. *EATCS Bulletin Vol. 81*, 2003.

[4] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Software and System Modeling*, 3(3), pages 194–209, 2004.

[5] Eclipse Consortium. *Eclipse Graphical Editing Framework (GEF 3.2)*, 2006. `http://www.eclipse.org/gef`.

[6] Eclipse Consortium. *Eclipse Modeling Framework (EMF 2.2)*, 2006. `http://www.eclipse.org/emf`.

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

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

[9] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, TU Berlin, Fak. IV, Books on Demand, 2006.

[10] C. Ermel, K. Ehrig, G. Taentzer, and E. Weiss. Object Oriented and Rule-based Design of Visual Languages using TIGER. *Proc. Graph-Based Tools (GraBaTs'06)*, ECEASST, 2006.

[11] Eclipse Consortium. *Eclipse Graphical Modeling Framework (GMF)*, 2007. `http://www.eclipse.org/gmf`.

[12] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3,4), pages 287–313, 1996.

[13] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11, pages 690–723, 1999.

[14] Seven Towns Limited. Rubik's Official Online Site, 2008. `http://www.rubiks.com/`.

[15] M. Minas. *DiaGen/DiaMeta – The Diagram Editor Generator*, 2007. `http://www.unibw.de/inf2/DiaGen/`.

[16] C. D. Pegden, R. E. Shannon, and R. P. Sadowski. *Introduction to Simulation Using SIMAN*. McGraw-Hill, 1990.

[17] E. Sugiyama, R. Osawa, and S.-H. Hong. Puzzle Generators and Symmetric Puzzle Layout. In *Proc. Asia Pacific Symposium on Information Visualisation (APVIS'05)*, pages 97–105, 2005.

[18] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, LNCS vol. 3062, pp. 446 – 456. Springer, 2004.

[19] Tiger Project Team, TU Berlin. *Tiger: Generating Visual Environments in Eclipse*, 2005. `http://www.tfs.cs.tu-berlin.de/tigerprj`.