Ludo: A Case Study for Graph Transformation Tools

Arend Rensink[1], Alexander Dotor[2], Claudia Ermel[3], Stefan Jurack[4], Ole Kniemeyer[5], Juan de Lara[6], Sonja Maier[7], Tom Staijen[8], and Albert Zündorf[9]

[1] Universiteit Twente, The Netherlands, rensink@cs.utwente.nl
[2] Universität Bayreuth, Germany, alexander.dotor@uni-bayreuth.de
[3] Technische Universität Berlin, Germany, Claudia.Ermel@tu-berlin.de
[4] Philipps-Universität Marburg, Germany, sjurack@Mathematik.Uni-Marburg.de
[5] BTU Cottbus, Germany, okn@informatik.tu-cottbus.de
[6] Universidad Autónoma de Madrid, Spain, juan.delara@uam.es
[7] Universität der Bundeswehr München, Germany, sonja.maier@unibw.de
[8] Universiteit Twente, The Netherlands, staijen@cs.utwente.nl
[9] Universität Kassel, Germany, zuendorf@uni-kassel.de

Abstract. In this paper we describe the *Ludo case*, one of the case studies of the AGTIVE 2007 Tool Contest (see [22]). After summarising the case description, we give an overview of the submitted solutions. In particular, we propose a number of dimensions along which choices had to be made when solving the case, essentially setting up a *solution space*; we then plot the spectrum of solutions actually encountered into this solution space. In addition, there is a brief description of the special features of each of the submissions, to do justice to those aspects that are not distinguished in the general solution space.

1 Introduction

This paper describes one of the three case studies chosen for the tool contest outlined in [22], based on a (children's) game that in English goes under the name *Ludo*. The motivation for choosing this case was that it provides the following tool challenges:

1. Modelling the rules of the game in an easy and understandable way;
2. Allowing the specification of different player strategies;
3. Simulating, storing and replaying different games in a flexible manner;
4. Visualising the game and allowing user interaction;
5. Offering high performance in simulating games.

The case was actually proposed by two different parties: Hölscher [14] and Kroll and Geiß [19], with somewhat different emphases: the former stresses the issue of different player strategies, the latter concentrates on some modelling aspects. The case descriptions are combined and summarised in Sect. 2 below.

The case received 8 solution submissions, with a fair diversity of approaches and choices for the different aspects of the case. In the remainder of this paper, after describing the case itself, in Sect. 3 we propose a number of dimensions or criteria along which one can distinguish solutions, and we match the submitted solutions against those criteria, thus setting up a *solution space*. Subsequently, Sect. 4 contains a short description

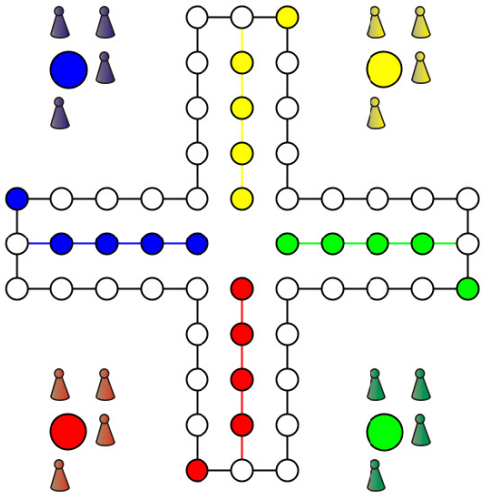

Fig. 1: The Ludo playing board

for each of the submitted solutions, highlighting those aspects that are insufficiently covered by the general criteria. Sect. 5 ends with a conclusion, evaluation and recommendation for future cases.

2 Ludo case description

In this section we describe the original case, by combining the original descriptions in [14, 19] and clearing up some ambiguities.

The goal of this case is to model the "Mensch ärgere dich nicht" game, the German variant of the Ludo game. The following is adapted from Wikipedia:

> "Mensch ärgere dich nicht" is a German board game, by Joseph Friedrich Schmidt (1907/1908). It is a Cross and Circle game, similar to the Indian game Pachisi, the American game Parcheesi, and the English game Ludo, though as with Ludo the circle is collapsed onto the cross.

2.1 The game

The Ludo board consists of a directed 40 field ring in form of a cross (see Fig. 1). The rules are as follows:

1. There are four *players*: traditionally, red, blue, yellow and green. Every player has four *pawns*, which are not in the game initially (they are "*at home*").
2. Every 10th field serves as *entry field* for a player. Note that this imposes a cyclic order over the players. In addition, directly preceding each entry field is a junction to four consecutive *goal fields* of the same player.
3. At every point in time, it is the *turn* of one of the players. Turns rotate according to the cyclic order of players.

4. The player whose turn it is throws a *six-sided die*, and moves one of his pawns according to one of the following rules, if any is applicable. If no rule is applicable, no pawn is moved.

   Entry: If the die shows a six and the player still has pawns at home, and the player's entry field is not already occupied by a pawn of his own, he must put one pawn from his home to his entry field.

   Forward: If no entry move is possible, the player must select one of his pawns on the board and move it forward by the exact number of eyes on the die. In doing so he may not pass (i.e., overtake) or end on his own entry field (instead he must take the junction to his goal fields) and may not end on a field that is already occupied by a pawn of his own. Moreover, a forward move may not pass any pawn already on a goal field.

   If there is already a pawn (of another player) on the target field of a move, then this pawn is *kicked* and returns to the other player's home.
5. If the die roll was a six, the same player takes another turn; otherwise, the next player (in the given order) gets his turn.

The game ends when one of the players has occupied all his goal fields. This player has won the game.

2.2 Strategies

As with any game, an interesting question from the point of view of formal analysis is to determine strategies for playing that are likely to win the game. Without going into game theory, for the particular case of Ludo one can easily identify several global strategies (global in the sense that they do not change during the game).

Aggressive: Give preference to a move that kicks a pawn;

Cautious: Give low priority to a move that kicks a pawn (so as not to anger the other player);

Defensive: Give preference to a move to a target field where the pawn cannot be kicked;

Move-first: Give preference to moving the foremost pawn;

Move-last: Give preference to moving the hindmost pawn.

More sophisticated strategies can be defined by taking the moves (or the strategies) of other players into account.

3 Solution space

In this section, we discuss some dimensions along which choices have to be made while modelling the game, and which therefore serve as a basis for distinguishing solutions. We end with a table in which all the solutions received are positioned along those dimensions.

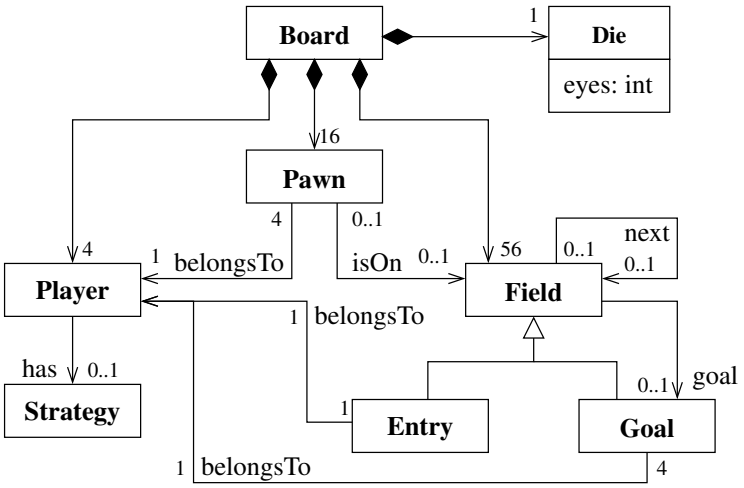

Fig. 2: Expalantory type graph for the Ludo case

3.1 Elements of the model

First of all, let us describe the essential elements of any graph transformation-based Ludo model. This comes down to selecting the concepts from the case description that are turned into node and edge types. The concepts are collectively displayed in Fig. 2 in a simple type graph. (Note that this was *not* part of the case description and does not necessarity have any connection with the type graphs or meta-models used in the solutions; it is just provided for explanatory purposes.)

Player. This is modelled by a node type. Players can have an identifier or colour to distinguish them. The cyclic order of players is typically modelled explicitly (through edges).

Pawn. This is modelled by a node type. Each pawn *belongs* to a certain player; this is typically modelled by an edge, or in some cases by an attribute.

Field. This is modelled by a node type. Entry and goal fields are special kinds of fields, typically modelled by subtypes, or in some cases marked by special self-loops. The same may be done for home fields, although these are not essential for the game (we left them out of Fig. 2). Entry and goal (and home) fields belong to a player; this is typically modelled by an edge. The position of a pawn (on a field) is likewise modelled by an edge.

Board. This consists of all the fields and their interconnections, i.e., the next fields and the junctions to the goal fields. The interconnections may be turned into nodes, but typically will be modelled by edges. The board itself does not necessarily need a special node, since all fields already implicitly belong to it.

Die. This is modelled by a (singleton) node, possibly of a special type but optionally integrated with some existing node. The outcome of a die roll is then typically an attribute of the die node.

Strategy. This is modelled by a node or an attribute value; a player can then be assigned a strategy by adding an edge or attribute.

3.2 Game rules

It is natural to turn the game rules into graph transformation rules. An important issue here is the *granularity* of the transformation rules: a rule can capture either a small part of a turn, on the level of a single step in the description of Sect. 2.1 or even smaller, or combine several such steps into an atomic whole.

The game rules impose restrictions in selecting the pawn to move, and also in executing the move. Some of these restrictions, such as the one that forbids passing a pawn on a goal field, are not straightforward to specify. An important choice is therefore whether the Ludo model indeed enforces all the game rules. There are at least the following four options:

- *A priori enforcement.* In this case, only moves that are according to the rules are ever enabled. This typically requires that the move itself is modelled by a single rule, which moves the pawn immediately to the target field.
- *A posteriori enforcement.* In this case, a move is tried out, and discarded if it leads to an illegal state, either by backtracking or marking the pawn als immovable. The actual move is then selected among the pawns that are not immovable.
- *No enforcement.* Depending on the underlying graph transformation tool, game rule enforcement may be out of scope altogether. In particular, this may be true of the solutions based on diagram editor generators: although they may offer complex editing operations that actually model a valid move, the simpler operations that result in "cheating" cannot always be turned off.

3.3 Modelling choices

The description above already indicates that there are a number of choices to be made in the model. We list the most interesting choice points and the possible options, below.

Randomness. Die rolls are supposed to be random, but graph transformation rule application is deterministic by design (once a rule match is established). It is therefore a choice point how to obtain the non-determinism, and even more difficult, the randomness needed here. On the other hand, the Ludo case is relatively benign in that there is an a priori fixed, small number of outcomes. (This would even allow an exhaustive enumeration of all possible outcomes using 6 different rules. However, none of the submitted solutions took this "ad hoc" approach.)

Options for implementing die rolls are:

- Calling a *system function* for a random number. This means that the graph transformation is not "pure" any more, but, on the other hand, randomness is guaranteed (insofar the underlying system guarantees it). The solution also works for more general random selections.
- *User query* for the outcome. Rather than asking the underlying system, a graph transformation rule may ask the user for a "random" value. This does result in non-determinism, but not in randomness (humans are notoriously bad at randomness). Since at the point of interaction all values are still possible, a case can be made that this solution is more "pure" than the first. It also works for more general random selections.

– *Match selection.* This solution also relies on human selection of a value, but here the potential outcomes are pre-determined as part of the start graph, resulting in six different matches; a choice among the resulting rule applications is offered to the user. This has the same disadvantages as the previous solution regarding randomness, and will only work as long as the number of values is finite (and preferably small); on the other hand, it falls entirely within the graph transformation formalism.
– *Random exploration.* In this solution, like the previous, all potential rule applications are pre-computed; one is then automatically chosen, as part of the state space exploration. In this case, randomness is once more guaranteed, but like the previous solution, it will only work if the outcomes can indeed be pre-determined.

In the solutions we have seen that the first option is favoured, whereas the last option also occurs once. The second and third do not occur.

Counting. The forward move involves counting fields. In other words, the length of the path that a pawn has to traverse is determined by a number in the graph itself, namely, the outcome of the die roll. There is a choice point in how to achieve this. Additional difficulties are: (i) the pawns must go to the goal fields rather than pass again to the entry field; and (ii) pawns on the goal field may not be overtaken. Leaving aside the obvious *ad hoc* solution of specifying one rule per die roll, which was (fortunately) not chosen in any of the solutions, viable options are:

– *Numbering the fields.* By numbering the fields consecutively, the target field of a pawn can be calculated by addition modulo the number of fields. In order to ensure the additional constraints, however, quantification is needed over the intermediate fields, which requires a more powerful notion of transformation rule.
– *Single-step rules.* The granularity of the rules can be made smaller, so that each rule application only moves the pawn by one step, at the same time decreasing a counter. There are then distinct rules for intermediate steps and for the last step (when the counter decreases to zero): only in the last case a test has to be included for the presence of pawns on the target field. Since the legality of a move can sometimes only be decided later on (for instance, a move is not legal if its final field is occupied by a pawn of the same player), this solution also requires some form of backtracking.
– *System functionality* for determining the correct target location. This means that the rules interact with the underlying system to invoke dedicated code; in other words, this part of the problem is not solved within the graph transformation formalism.

Strategies. To implement a player strategy, one has to select between allowed moves on the basis of a ranking: first try out the best (kind of) move, then (if that is not possible) a less preferable one, etc. Ideally, this selection should be orthogonal to the moves themselves, i.e., the rules describing the moves should not have to be adapted in order to take strategies into account. This, however, is not easy to realise, given the fact that the strategies impose a complex ranking. In fact, there are two types of ranking: position-based and result-based.

The foremost and hindmost strategies are position-based, in that they select a move on the basis of the position of the pawn that moves. Note that it is not enough to simply require that the fore- or hindmost pawn must move, since if this pawn *cannot* move (because one of the other constraints would be violated) then the next one (from the front or back) should be selected instead, and so forth.

The aggressive, cautious and defensive strategies are result-based, in that they select a move on the basis of the outcome. This is in a sense easier than the former type of ranking, since such a condition on the outcome is essentially a right application condition in the rule, which can typically also be translated to a left application condition. In combination with rule priorities or some other form of control, this has the desired effect.

3.4 Graph formalism

Regarding the graph transformation formalism, we distinguish the following dimensions of choice. (Note that these choices are made on the level of the graph transformation tool, and not the Ludo model.)

- The *typing* available for the graphs. All but one of the tools have a built-in notion of typing, which usually is given in the form of a type graph. In some cases these type graphs conform to an existing (standardised) meta-model, namely EMF. In one case the typing is actually determined by the underlying programming language.
- The *language* in which the rules are formulated. For most (in fact all but one) submissions this is a visual format; only one submission requires a textual input of the rules. If rules are specified visually, there is still a choice between the abstract graph or concrete syntax level; see Sect. 3.5 below.
- The *control* that is imposed on top of the graph transformation rules. The amount of control that a tool offers is an important factor in the ease with which complex game rules can be easily specified and enforced (see above). Control can range from none to a full-fledged language in which rule applications can be specified, including hints about their matchings. An intermediate option is *prioritised*, meaning that the rules have fixed global priorities. In practice we have encountered two kinds of control languages: *imperative* (programming language-style) and *storyboarded*, which is the FuJaBa speciality (see Sect. 4.1).

3.5 Visualisation

From a "lay user's" (rather than a tool developer's) perspective, one of the most important features of a graph transformation tool is surely its ability to show the graphs in a nice, easily comprehensible manner. There is a wide range of capabilities among the submitted solutions.

- *Plain graphs.* The base level, which all tools offer, is to show the abstract graphs that constitute the model. This means, for instance, that the order of the players, the numbering of the fields, etcetera, which are only there for the model and do not provide useful information for the game player, are nevertheless visible. Typically,

moreover, on this level no extensive layouting support is available — and even if available, the layout information is not considered to be part of the model.
- *Concrete syntax.* A much more sophisticated visualisation is achieved if a concrete, domain-specific syntax can be defined on top of the abstract graphs. This makes for solutions that really offer something looking like a Ludo game board.
- *3D Rendering.* By far the most attractive visualisation, which only one of the solutions can offer, is a 3D view of the board. This requires a rendering mechanism that is much more sophisticated even than the concrete syntax solution described above.

3.6 Interaction

The unit of interaction between user and Ludo model is in principle a single rule application — which is indeed the obvious choice given the setting of graph transformation. However, the way applications are selected can differ, as well as the degree to which rule selection can be automated. Possible options are:

- *GUI-based interaction.* If the visualisation offers a concrete, Ludo-specific GUI view, then it may also offer functionality for selecting moves by interacting directly with this view, meaning that the rules become completely invisible. In other words, the model can have the look-and-feel of a mature game application.
- *Match selection.* Most of the tools work on the basis of pre-computed matches. The interaction is then typically through a user-guided selection of the rule to be applied, including the match if there is more than one (which is the case if there is more than one pawn that can move, or in some cases also in order to select the die roll, see Sect. 3.3).
- *Match construction.* For tools that do not rely on pre-computed matches, the user must *construct* the match by hand. A rule is executed once a legal match has been selected.
- *Partially automatic.* If there is only a single applicable rule, and the tool is able to detect this (meaning that it does not rely on user-guided match construction), then there is the possibility of executing this rule straight away, without requiring user interaction. Alternatively, some rules may always be executed automatically, whereas others (the human player's moves) always wait for user input.
- *Fully automatic.* A further step towards automation consists of automatic rule selection and execution even in the case of non-determinism. This means that a tool can play a game all on its own, without user interaction.

3.7 Analysis

A final choice point in the solutions is the amount of analysis that has been done regarding different player strategies. In particular, by letting different strategies play against one another, one may attempt to determine the best strategy experimentally. For this to be possible, the tool must first of all support fully automatic game play (see Sect. 3.6), and secondly have a performance good enough to play a reasonable number of games.

Table 3: Solution space

		FuJaBa	FuJaBa/GMF	DiaMeta	XL	AGG/ROOTS	AToM3	Groove	Tiger	
Game rules	A priori			X	X		X	X	X	
	A posteriori	X				X				
	Cheats possible		X	X					X	
	Granularity	T	T	P	T	P	P	P	S	Turn / Phase / Small step
Randomness	System function	X	X	X	X	X	X		X	
	Exploration							X		
Counting	Numbered fields					X	X	X		
	Small steps	X	X	X					X	
	System function				X		X			
Strategies	Position-based	X		X	X	X	X			
	Result-based	X		X	X	X	X	X		
Formalism	Typing	T	M	M	P	T	T		T	Type graph / Metamodel / Program types
	Rule language	A	A	A	T	A	C	A	C	Concrete visual / Abstract visual / Textual
	Control	S	S	I	I		P	P		Prioritised / Storyboarded / Imperative
Visualisation	Rendered				X					
	Concrete syntax	X	GMF	X			Python		X	
	Abstract syntax				X	X		X		
Interaction	GUI-based	X		X	X					
	Match selection					X	X	X		
	Match construction		X						X	
	Partially automatic					X	X			
	Fully automatic	X	X	X	X	X		X		
Analysis	Performance	50	50	2600	290			1500		ms/game, rough average
	Experiments	X						X		

3.8 Overview

In Table 3 we show the resulting table of choice points for the solutions received; see also [8, 10, 20, 17, 15, 5, 2, 1].

4 Individual solutions

4.1 Fujaba

At the University of Kassel we use the Ludo game as an exercise for our courses in object oriented modeling with Fujaba (see [13]) for about 4 years now. We have also used it within highschool courses in computer science as an example for beginners. Thus, we have many experiences with this example and it was easy for us to come up with a case study for the Agtive tool contest. Our case study addresses all the topics mentioned in the Ludo tool contest: we have modeled the game rules. We have developed a graphical user interface for interactive playing. For this contest, we have developed automatic player strategies and a driver for automatic simulations. Note, seeding our random number generator results in deterministic game simulations.

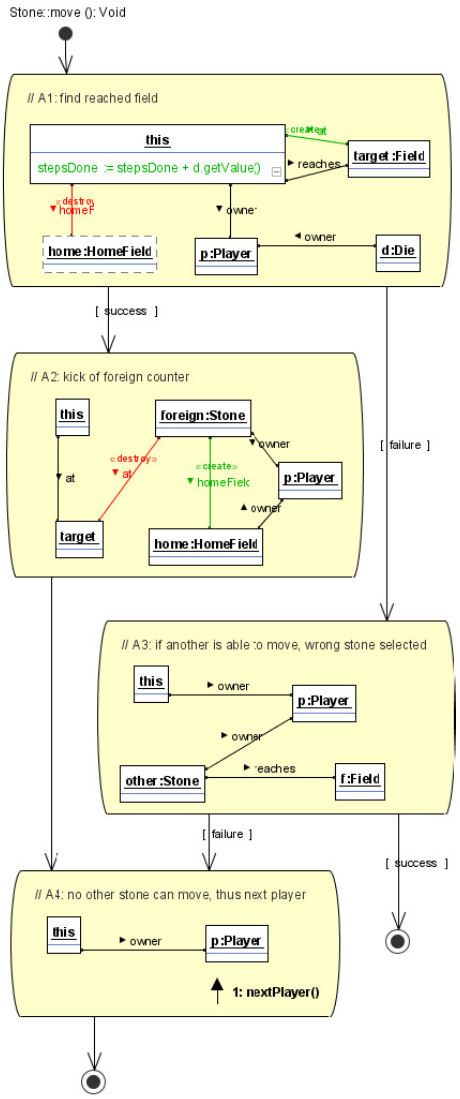

Fig. 4: Example storyboard

The first part of the challenge is the modelling of the rules of the game. For instance, Fig. 4 shows the `move` method of class `Stone` which is invoked when the user clicks on a pawn during the game. This kind of diagram is called a *storyboard*. The activity comment starts with an identifer that we use for reference. Activity A1 uses a `reaches` link to look up the target field that is reached in the current situation. Note, if method `getReaches` returns `null`, this lookup fails and accordingly, activity A1 would fail. The rest of the storyboard implements the complete move. As a simple GUI framework we use the WhiteSocks library, cf. [9]; see Fig. 5. This is created by turning the Ludo model elements into WS objects, and by assigning appropriate icons and labels. One may play the game by clicking on the die; this will compute a new

Fig. 5: Ludo game built with Fujaba and Whitesocks

die value and update the die icon, accordingly. Then, the player may click on one of his pawns. This will move the icon above the reached field and the die will be forwarded to the next player.

To simulate and rerun games, one may just store the start situation and then start the game with automatic players on. If one seeds the die correctly, the game will rerun similar to previous runs with the same seed.

As an example for a simulation, we have run 100 games with 2 level 7 automatic players positioned at 12 and at 3 o'clock at the board. As expected, the player at 3 o'clock has a little disadvantage because pawns waiting at the entry field of the 12 o'clock player may kick his pawns just before they enter the last lane before its goal fields. However, it was 57 wins for the 12 o'clock player and 43 for the 3 o'clock player. To simulate one game we need about 50 milliseconds where 60% of the time is devoted to the computation of priorities for the automatic players.

From our point of view, Fujaba is well suited for modelling the rules of the Ludo game and for the development of automatic player strategies. With the help of the WhiteSocks framework, it was easy to build a graphical user interface for the game. There may be multiple human and or automatic players at one computer or with the help of the Coobra environment mutliple player may play over the net. While the simulation performance is reasonable, we have once again recognized that our intensive usage of Java exceptions is a bottleneck for the generated code. We plan to improve this soon.

4.2 Fujaba and GMF

The following solution uses Fujaba [13] as well as the Eclipse Modelling Framework (EMF) [11] and the Graphical Modelling Framework (GMF) [12] to generate an automatic Ludo player and a Ludo editor to create, display and play the game.

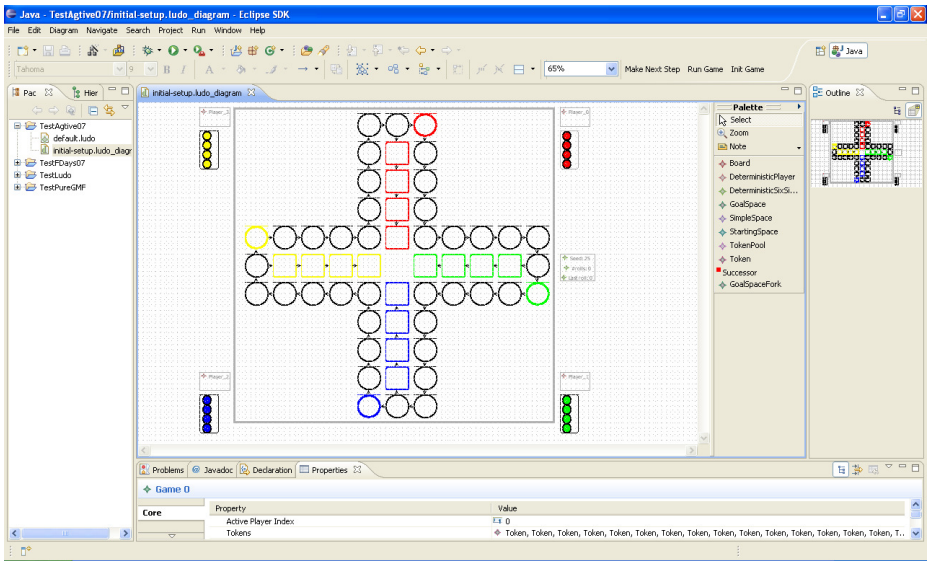

Fig. 6: GMF Ludo editor with initial board setup

First the structure of Ludo is modelled as UML class diagram. Second the behaviour is modeled with story diagrams, a combination of activity and communication diagrams, from which Fujaba is able to generate executable code. Fujaba is able to map the Fujaba-Metamodel onto Ecore and to inject the story diagram based methods into the EMF code generation [3]. The result is Ecore-compliant executable code which serves as input for the GMF to generate a Ludo editor [4]. This editor is used to create the initial board setup (see Fig. 6).

The basic editor commands allow playing Ludo but they do little to enforce valid moves. Buttons and context menus are added to execute the story diagram based methods which allow valid moves only. Furthermore the figures of colored game elements have to be enhanced in order to color them in dependency of their owner [12]. Both enhancements do not require manipulation of the generated code but are loaded in a separate plugin.

Highlights: Fujaba allows to model the behavior graphically in story diagrams which increases the readability. The generation of the editor by GMF reduces the implementation of a sophisticated GUI tremendously. As GMF is designed for extensibility the editor can be enhanced easily. As an Eclipse plugin the solution can be deployed platform independently. See [4] for more details.

Open issues: The Ludo editor intermingles both editing the game board and playing the game, so it is possible to cheat by editing the board during play. Also the mapping between Ecore-model and graphical model is limited and requires manual coding (in case of the colored elements). The missing backward trace from compiler (and runtime) errors to Fujaba diagram elements makes debugging a tedious task.

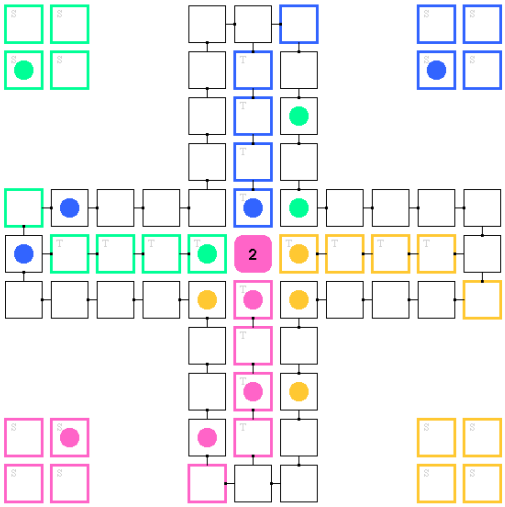

Fig. 7: Ludo board created with DiaMeta

4.3 DiaMeta

We used the diagram editor generator framework DIAMETA [21] to specify the board game Ludo. The generated editor offers the possibility to specify a board and to play the game.

Specification. To create an editor for a specific diagram language with DIAMETA [21], the editor developer has to provide two specifications: First, the abstract syntax of the diagram language in terms of its model (EMF model). Second, the Designer specification that primarily contains the visual appearance of diagram components, the interaction specification and the structured editing operations. Additionally, a layouter had to be programmed since DiaMeta does not yet support automatically generating a layouter.

Functionality. The generated editor makes it easy to create different boards, e.g., varying the number of fields or pawns. Fig. 7 shows a board that was created with the editor. A board consists of a die and some connected fields. For each player, we need a certain number of pawns, home fields, entry fields and goal fields.

To play the game, the editor offers two possibilities: Having a human player that rolls a die and then moves a pawn by hand, or choosing a strategy to play the game automatically.

A human player can either operate in free-hand editing mode, or in structured editing mode. In the first case, the editor user rolls the die and then grabs a pawn with the mouse and moves it somewhere on the board. It is not checked whether the move is allowed or not. In the second case, he rolls the die and then moves the pawn by clicking a button. In this case, it is checked whether the move is allowed or not.

Besides that we have the option to use a strategy, either for a single step, or to play the complete game. We can move the pawn that is nearest to the goal fields (Move-last), or we can move the pawn that is farthest from the goal fields (Move-first). Another

criterion is to choose whether a player always tries to kick other players' pawns when possible (Aggressive), or if the player only kicks other players' pawns if left with no other choice (Cautious).

Challenges. Most of the editor specification was easy to write. Two parts were challenging:

- First, DIAMETA uses a very simple language to specify structured editing rules and operations that have been used to specify strategy.
- Second, we had to write some parts by hand: the visualization of some components and accessors to attributes that are used in the interaction specification. Fortunately, DIAMETA offers the possibility to include self-written code, and hence made it easy to complete the editor.

4.4 Solution Using XL and GroIMP

This solution benefits from several XL features like iterated patterns (subsets of transitive closures) and optional patterns, and from the built-in 3D visualization and interaction of GroIMP [18]. Assuming that our graph of fields has the suggested structure of the Karlsruhe case study [19] (i.e., the edges between fields indicate the legal paths for each individual player), and that pawns of each player form a circular list linked by `next` edges, the pawn movement for the Karlsruhe variant can be implemented by a single rule:

```
(* d:Die < p:Player *) -tryNext-> (* Pawn *) (-next->){0,3}:
(
    (* f:Pawn [-next-> n:Pawn] *)
    <+ (* a:Field *) (-edges[p.index]->){d.score()} b:Field
    (? +> g:Pawn(gp,h)), ((g == null) || (gp != p))
)
==>> b [f], if(g != null) (h [g]), p -tryNext-> n;
```

It makes use of an iterated pattern `(-next->){0,3} : (...)` which traverses 0 to 3 `next` edges (but as few as possible) to find the actually moved pawn `f`, starting at the pawn indicated by a `tryNext` edge. The second iterated pattern `(-edges[p.index]->){d.score()}` traverses exactly as many edges of the distinct edge type of the player as the score prescribes (where the implementation of the random number generator is very easy as XL extends Java). The optional pattern `(? +> g:Pawn(gp,h))` tests if there is some other pawn on the potential new field `b`. For a match of the whole left-hand side, i.e., if there is a legal move, the rule is applied and moves `f` to `b`, `g` to its base field, and marks the next player to be tried. On a 3 GHz computer using an initial seed of 98754321, the complete sequence of 460 moves takes about 290 milliseconds.

A visualization can be obtained easily within GroIMP by using predefined geometric classes as superclasses for our nodes:

```
module Field extends Cylinder(0.001, 0.4);
```

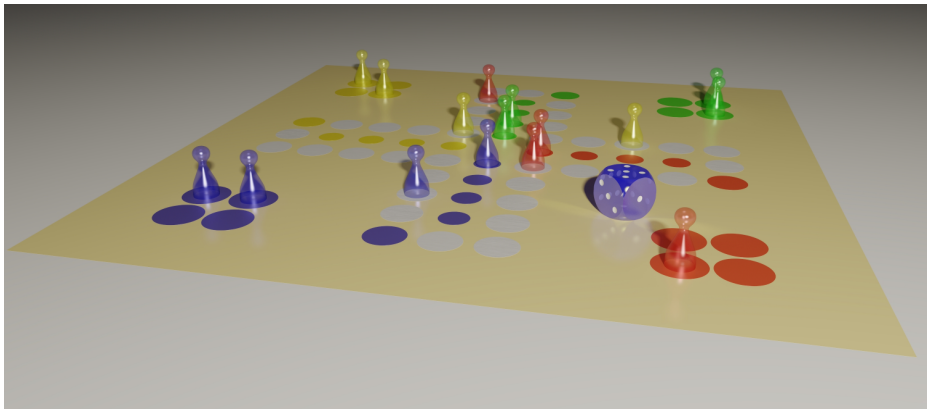

Fig. 8: 3D Visualization using GroIMP, rendered by integrated raytracer

For the pawns, we may also use an interactively modelled spline curve to create a surface of revolution. As each field defines its own local coordinate system, a pawn is automatically moved in 3D space from one field to another by simply redirecting its incoming edge as it is done by the movement rule. If we interactively design shaders (definitions of optical properties), we arrive at Fig. 8.

The extension of the rules to the complete set of Sect. 2.1 was also done with the exception of the interdiction to pass pawns at goal fields (but the latter could be integrated as a condition in the iterated pattern). Likewise, several strategies as well as human players were implemented. The latter can be controlled by hot-keys, but a mouse-based selection of the pawn to be moved would be possible without great effort, too.

4.5 ROOTS

The Rule-based Object-oriented Transformation System (ROOTS) is a plug-in for Eclipse, which is based on the graph transformation engine AGG following the algebraic approach. For further information on this tool see [16]. The basis of this Ludo implementation is a type graph including all elements of the game, e.g., pawn, die, fields, strategies etc. These are represented in an object-oriented manner i.e. by attributed classes, associations and inheritance. The virtual game board (clipping shown on the left of Fig. 9) is constituted by an instance of this type graph arranged analogously to the original board layout. In contrast to other solutions presented in this volume, ROOTS does not generate/compile any concrete syntax editors but directly shows the abstract syntax and allows detailed tracing of graph transformation steps.

The implemented rules define the game rules. They can be distinguished according to three different concerns: (1) starting phase e.g. negotiating the first player, (2) general game play e.g. moving a pawn, and (3) strategy-specific decisions (e.g., which pawn to move). The rule ‘Roll Die’ related to the first concern is exemplarily depicted on the right of Fig. 9. It demonstrates the capability of exploiting Java expressions, in particular in this case to throw the die at random. Since the strategy-relevant decisions are separated, common operations benefit from reuse and flexibility in strategy usage (during game play), by simply associating a strategy object (cp. game board graph in

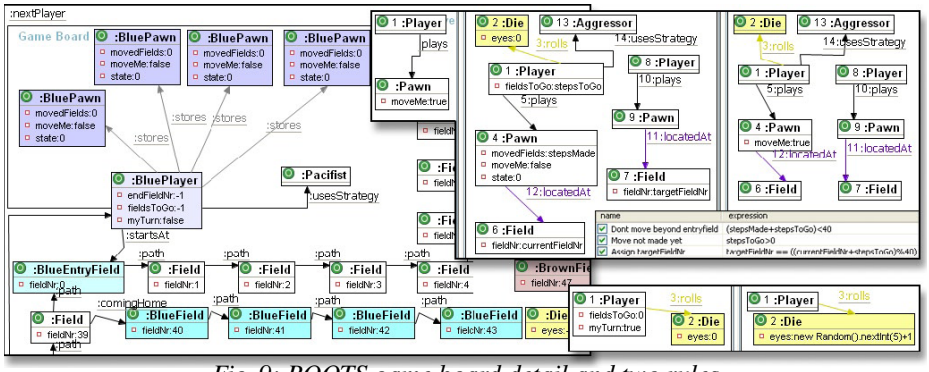

Fig. 9: ROOTS game board detail and two rules

Fig. 9), or even omitted enabling a human player. Four different automated strategies are realized: Pacifist, Shepherd, Runner and Aggressor. The rule 'Aggressor: Mark valid move' is shown in the right corner of Fig. 9.

Our solution is purely rule-based, i.e. we use graph elements and especially attributes to control the application of rules. To support a good understanding ROOTS provides the possibility to put descriptions on almost every element.

4.6 AToM^3

AToM^3 [7] is a tool for the generation of modelling environments for Domain Specific Visual Languages (DSVLs). It allows describing the DSVL abstract syntax by means of a meta-model with constraints (specified in Python). The concrete syntax is specified by assigning icon-like entities to classes and arrow-like elements to associations. It is possible to define model manipulations by graph transformations. These can be customized to work under the double or single pushout approaches [23]. Rules may use the inheritance relations defined in the meta-model [6], can have application conditions of the form $p \rightarrow q$ and have a priority, so that these with higher priority are tried first. Transformations can be executed in interactive mode (the user chooses the match), or batch (rules are executed until the grammar finishes). A delay can be assigned to the rules so that the rule execution can be animated. Starting from the meta-model and the concrete syntax specifications, AToM^3 generates a customized modelling environment for the DSVL. The user interface of the environment is also a model, and can be customized, e.g. adding buttons to execute transformations.

A generalization of the Spanish Ludo (called *Parchis*) has been modelled, allowing some degree of parameterization regarding the board topology, and the number of: players and their colours, pawns per player, fields to be counted when kicking pawns, and when a pawn reaches the finish. The resulting environment is shown in Fig. 10.

The game dynamics were specified using Double Pushout rules. A button was added in the final user interface to execute the transformation. The grammar runs in interactive mode, so the user selects a match for a rule if more than one is available (thus he selects the pawn to be moved). The rules moving the pawns of computer players usually produce unique matches, so no decisions have to be made (however sometimes two "equivalent" moves have to be chosen, e.g., when two pawns are the first ones, or when

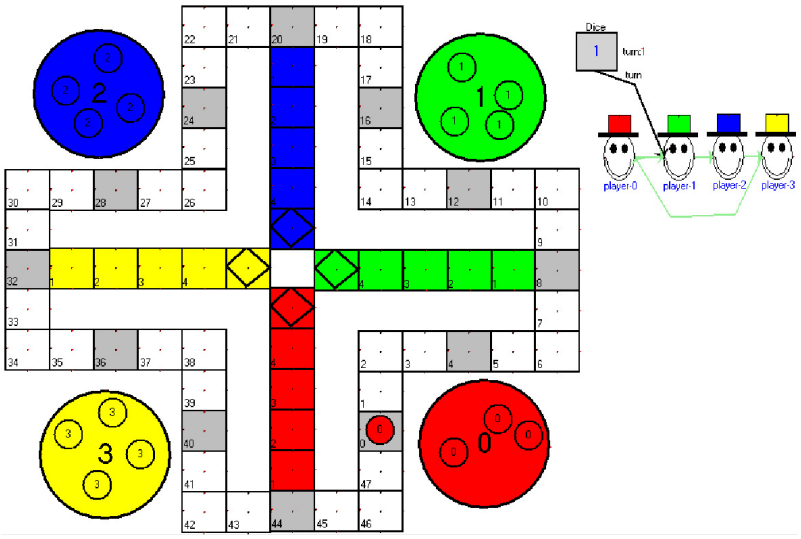

Fig. 10: The AToM^3 Generated Environment.

several pawns can be eaten). Regarding the visualization, rules moving pawns take care of placing them inside the target cells by means of Python code.

4.7 GROOVE

Our solution of the Ludo case is a specification of the game using the Groove tool set. Groove is a tool for graph transformations that uses directed, edge labelled simple graphs and the SPO approach. Given a graph grammar (G, P), composed of a start graph G and a set of production rules P, the tool allows to compute a labelled transition system (LTS) corresponding to all possible derivations in this grammar.

For the Ludo case, a graph is specified that models the Ludo board and the four players with their pawns. The actions of the players, including the constraints imposed by the rules of the game, are modelled as a set of nine graph transformation rules. These rules are applied in four steps: rolling the die, moving a pawn, kicking another players' pawn and selecting the player to have the next turn.

While modelling the game, we tried to keep the graph as simple and straightforward as possible (for both memory and visualisation reasons) while still being able to specify pawn-movement in a single rule, to minimize the size of the generated transition system. This is achieved by flagging the player nodes with a colour. Fields are either connected by *next* edges or by edges labelled with these colours, indicating which players are allowed to move between the fields. Groove's feature to match regular expressions (over labels on edges connecting nodes) allows to simply specify rules that move pawns into the players home and that disallow pawns to stay on the board more then a single lap.

One of the challenges was to have random results for rolling a die. The *die-roll* rule always has six possible derivations: one for each possible value of the die. We use Groove's *barbed exploration strategy* to achieve randomness. For a given state in the LTS,

this strategy determines all possible rule applications and adds them (and the resulting target graphs) to the LTS. It then randomly selects one of the unexplored target graphs (through a random generator built into the barbed strategy) and continues the barbed exploration from that graph. This is shown in Fig. 11, which displays a fragment of the explored part of the LTS. The "broomsticks" where 6 possible die rolls are evaluated are clearly visible, as is the fact that only one of the choices is eventually taken.

Simulation of the grammar in Groove generates an LTS in which each path represents a possible Ludo game. The barbed strategy typically explores one path of the full LTS until a final graph — whenever a player has won the game — has been found. We found that these paths often start with a cycle, representing a round where none of the players have thrown a six yet, and thus end in the same graph as they started.

Player strategies are implemented by adding strategy-rules with a higher priority than the *move* rule, replacing it in specific cases. Example strategies implemented in this way are *foremost* and *aggressive* as discussed in Sect. 2.2. To apply a strategy to a player, the *Player* node can be tagged with the name of the strategy, which is required by the rule. This allows different players to use distinct strategies.

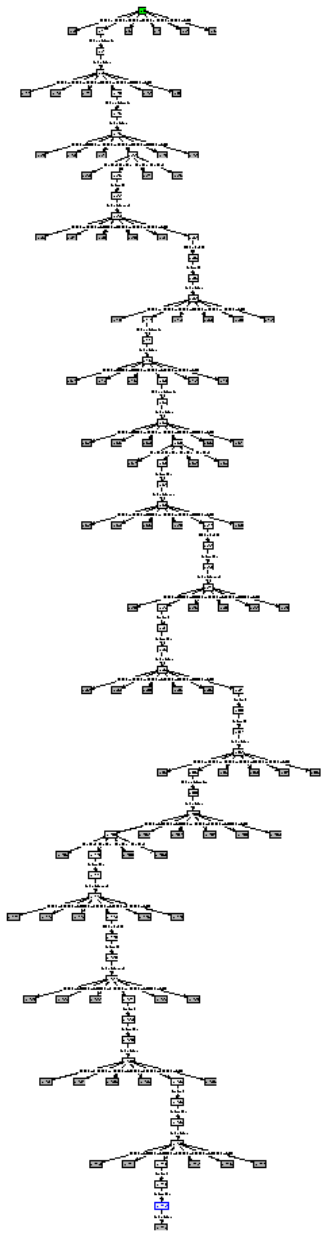

Fig. 11: Partial state space in GROOVE

4.8 Tiger plays Ludo

The TIGER project (transformation-based generation of environments) [24] aims at generating visual editors as plug-ins for ECLIPSE from a visual language specification which is based on a typed graph grammar. The TIGER *Designer* allows a language designer to define concrete visual graphics for language elements and to use this concrete syntax to define editor operations as graph rules.

The TIGER *Generator* generates visual editors where all defined editor operations are provided in the palette. In order to perform an editor operation (e.g. insert a symbol), the user has to select a rule from the palette, and, if required, to click on match objects in the editor panel where the rule should be applied. Since editor usage is highly interactive (i.e. each editor operation is an action evoked by the user), TIGER does not provide means to control rule applications.

In our Ludo specification, the board with tokens in their initial position was defined as a start graph. Each graph rule for game simulation represents a phase in the game, like selecting the first player, throwing the die, moving forward, or kicking out another player's token (see the rule palette of the generated Ludo tool to the right). The only strategic choices we allow the player to make are the following: who will be the first player of the game, which token shall move (in case there is more than one token of the player's color on the field), and which token shall go to the start place (if a six has been thrown). Hence, strategies are interactive user decisions, e.g., selecting from different applicable rules or choosing one of several possible matches. In the cases where there is no choice left, only one rule will be applicable at one match to go on with the game. Due to TIGER's nature, this rule still has to be selected from the palette instead of being applied automatically.

Specifying Ludo using the TIGER Designer, rules are edited using the concrete syntax, see e.g. rule *moveOneStep* in Fig. 12 (a). Note that this rule has a set of negative application conditions (not depicted), forbidding e.g. that the next field is the entry field of the active player. Fig. 12 (b) shows the generated Ludo game environment with the Ludo board in the editor panel besides the game rule palette. The four colored fields around the die control the turns of the players. The current player and his currently selected pawn are marked by colored rings around the respective fields. Please note that

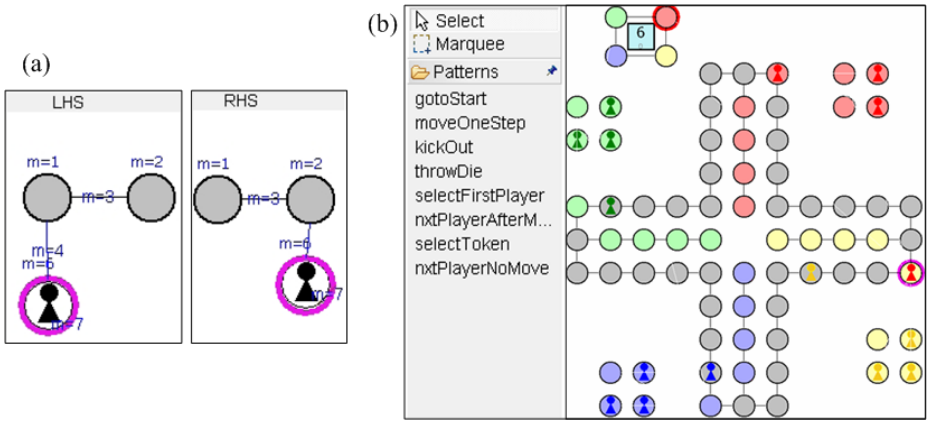

Fig. 12: TIGER *rule* moveOneStep *(a) and Ludo Game Environment (b)*

due to TIGER being an editor generator, the palette may easily be extended by editing rules for drawing the game board, thus adding a flexibility of designing user-specific Ludo boards as part of the game simulation.

5 Conclusions

The response to the Ludo case has been a quite diverse set of solutions. We refer again to Table 3 for an overview and comparison along the established choice points. This multitude of solutions is, of course, very positive: clearly, many tool developers have been

challenged to show what they can achieve when modelling this well-known application. Indeed, the Ludo case descriptions have left a lot of room for different interpretations and special features.

The same observation also has a negative connotation: given this diversity, there is no very objective basis for comparing, let alone ranking, the submissions. Although in Table 3 we did manage to set up a number of "choice dimensions", largely inspired by the solutions that we actually received, it would in fact be preferable to identify beforehand what the expected modelling challenges are, and in some cases perhaps also how we want them to be addressed. An example of one aspect that, in our opinion, could have been worked out to greater effect, is the analysis of the player strategies.

We recommend that a next tool contest again includes a case that is essentially about modelling a system, with at least the complexity of the Ludo game. In fact, we would favour another game-related application, since, as we have seen, this offers scope for many different tool approaches. However, we also recommend that a list of case aspects is provided beforehand, with for each aspect a description of what should be addressed. Example aspects can be found among the choice dimensions in Table 3:

- *Modelling.* This concerns particular game characteristics that are expected to be hard to model.
- *Analysis.* This concerns investigating actual game runs, comparing player strategies, etc. Possibly some performance criteria could be identified. Alternatively, correctness issues such as termination may be identified.
- *Visualisation.* This concerns creating an appealing or understandable visual model environment for the game application.
- *Interaction.* This is about creating a playable game. It might be worthwhile trying to get different graph transformation engines to play against one another.
- *Other.* In order to prevent restricting the creativity of submitters, the list of aspects should not be closed.

Submitters can select those case aspects that they will concentrate on. In this way each submission can display its own strengths, on the basis of a common, shared application, and yet comparisons can be made, along lines that were set out and known beforehand. Thus, the advantages of Ludo are kept, but we will be able to draw more value out of it.

References

[1] E. Biermann and C. Ermel. Tiger plays Ludo. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[2] I. Boneva, H. Kastenberg, T. Staijen, and A. Rensink. The Ludo Game with the Groove Tool Set. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[3] T. Buchmann, A. Dotor, and L. Geiger. Emf codegeneration with fujaba. Submitted to the FujabaDays'07 conference., 2007.

[4] T. Buchmann, A. Dotor, and B. Westfechtel. Model driven development of graphical tools: Fujaba meets gmf. In *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 425–430. INSTICC, July 2007.

[5] J. de Lara. Generating a Tool to Play Ludo with AToM^3. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[6] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376:139–163, 2007.

[7] J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism modelling and meta-modelling. In *FASE '02*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[8] I. Diethelm, L. Geiger, and A. Zndorf. Implementing Ludo with Fujaba. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[9] I. Diethelm, R. Jubeh, A. Koch, and A. Zündorf. Whitesocks - a simple GUI framework for Fujaba. In *International FujabaDays 2007, Kassel, Germany*, 2007.

[10] A. Dotor and T. Buchmann. Building Ludo with Fujaba and the Graphical Modeling Framework (GMF). URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[11] Eclipse Foundation. *The Eclipse Modeling Framework (EMF) Overview*, 2005. http://www.eclipse.org/linebreak[0]modeling/emf.

[12] Eclipse Foundation. *GMF - Graphical Modeling Framework*, 2006. www.eclipse.org/gmf.

[13] The fujaba toolsuite. URL: www.fujaba.de, 2006.

[14] K. Hölscher. Case proposal: Don't get angry. URL: http://gtcases.cs.utwente.nl/wiki/uploads/ludo_bremen.pdf, 2007.

[15] S. Jurack and G. Taentzer. Realizing Ludo by ROOTS. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[16] S. Jurack and G. Taentzer. ROOTS: An Eclipse Plug-in for Graph Transoformation Systems based on AGG. In *This volume*, 2008.

[17] O. Kniemeyer. Ludo — Solution using XL. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[18] O. Kniemeyer and W. Kurth. The modelling platform GroIMP and the programming language XL. In *This volume*, 2008.

[19] M. Kroll and R. Geiß. A Ludo Board Game for the AGTIVE 2007 Tool Contest. URL: http://gtcases.cs.utwente.nl/wiki/uploads/ludo_karlsruhe.pdf, 2007.

[20] S. Maier and M. Minas. Ludo meets DiaMeta. URL: http://gtcases.cs.utwente.nl/wiki/Ludo, 2007.

[21] M. Minas. Generating meta-model-based freehand editors. Electronic Communications of the EASST, Proc. of 3rd Intl. Workshop on Graph Based Tools, 2006.

[22] A. Rensink and G. Taentzer. AGTIVE 2007 graph transformation tool contest. In *This volume*, 2008.

[23] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific., 1997.

[24] Tiger Project Team, Technical University of Berlin. *Tiger: Generating Visual Environments in Eclipse*, 2005. http://www.tfs.cs.tu-berlin.de/tigerprj.