Evaluation of the Story Driven Modeling Methodology: From Towers to Models

Article · January 2007

CITATIONS
0

READS
32

1 author:

Pieter Van Gorp
Eindhoven University of Technology
89 PUBLICATIONS **1,619** CITATIONS

SEE PROFILE

Evaluation of the Story Driven Modeling Methodology: From *Towers* to *Models*

Pieter Van Gorp
University of Antwerp, Belgium
pieter.vangorp@ua.ac.be

***Abstract*—Story Driven Modeling (SDM [1], [2]) aims to provide a systematic approach to turn requirements scenario's in class structures and method implementations. It extends noun-verb analysis [3], Object Role Modeling (ORM [4]) and CRC cards [5] both notationally and methodologically. More specifically, it provides a notation for recording use case scenario's and a derivation strategy for generalizing scenario descriptions (called "Story Boards") into general method implementations (called "Story Diagrams"). This paper critically evaluates this methodology by extending the *Towers of Hanoi* solution described by Diethelm et al. [6], [7], [8]. Moreover, the paper motivates why the methodology is a promising basis for developers of model transformations.**

This paper is structured as follows: Section I articulates the use case scenario's of the Towers of Hanoi problem as a set of Story Boards. These scenario's are gradually generalized into the Story Diagrams from Section II. This section also discusses the graph transfomation fundamentals in general. Moreover, it motivates why control structures are essential by briefly discussing the lessons learned from a case study in which a pure graph grammatical approach has been applied. Section III draws conclusions from these modeling exercises.

I. Story Boards

This section presents an extension of the example from Diethelm et al. [9] to illustrate the syntax for story boarding. More specifically, subsection I-A presents a solution to the well-known problem of the Towers of Hanoi to illustrate how system snapshots can be visualized in the context of use case scenario's. Subsections I-B and I-C illustrate the process for turning such specific scenario descriptions into executable behavioral models. Finally, subsection I-D critically evaluates the steps performed. All models related to the *Towers of Hanoi* problem have been developed with the Fujaba tool [10].

A. Educational Sample: Towers of Hanoi

Figure 1 presents the evolution of moving one disk from the left to the right tower. The disk starts on the left tower (State 1). In State 1, it is removed from this tower, as specified by the ≪*destroy*≫ construct, and added to the right tower, as specified by the ≪*create*≫ construct. The scenario ends as soon as the disk is properly located on the target tower (State 3).

Note that this diagram intentionally models a very specific case, as this is most comprehensible to domain experts. Also note that the visual syntax enables one to arrange elements in a layout that makes sense in the problem domain: the three towers are arranged next to each other and disks are arranged on top. For conciseness, the following Story Boards will not represent the *algorithm* object with a custom icon anymore.

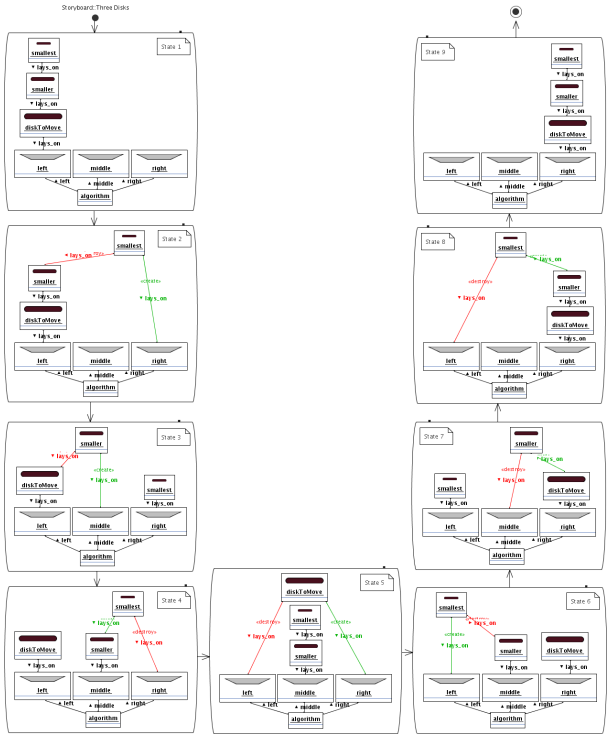


Figure 3. Towers of Hanoi – Scenario for 3 disks.

Figure 2 models how two disks are moved from the start tower to the target tower. It enables one to analyze this extended case carefully and discuss it with other developers. In State 2, the smaller disk is moved to the auxiliary tower "middle" such that in State 3, the taller disk can be moved to the target tower. In State 4, the smaller disk is moved on top of the taller one. The scenario ends after State 5, when the "disk to move" and all smaller disks on top of it have been moved to the right tower. This model is still scenario-specific and can be compared with an explicit representation of real-life events described by a domain expert.

Finally, Figure 3 models the use case scenario for moving three disks from the left tower to the right one. After creating or reading this model, one may have discovered the recurring behavior that can be implemented using the famous

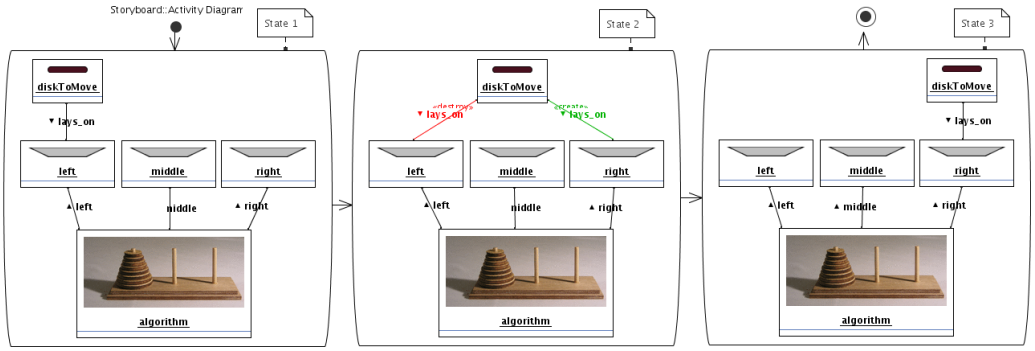


Figure 1. Towers of Hanoi – Scenario for 1 disk.

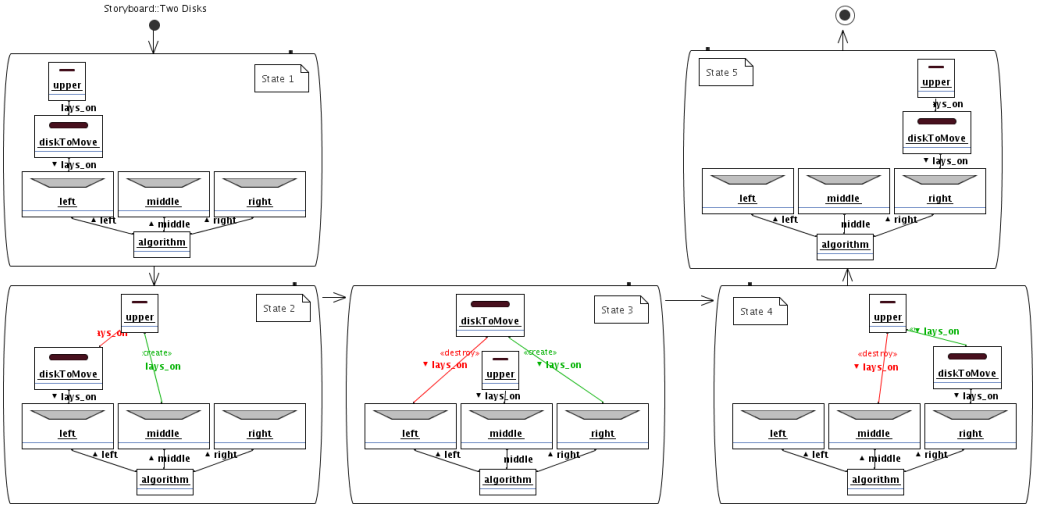


Figure 2. Towers of Hanoi – Scenario for 2 disks.

recursive solution to the problem. However, SDM does not force developers to invent the correct method implementation directly. Instead, it offers a set of guidelines for systematically transforming the above scenario's in a syntactically similar, yet more abstract and compact Story Diagram. While we refer to [9] specifically and [7], [8] in general for the details of the derivation algorithm, the following sections will motivate its relevance and briefly illustrate its application by means of the running example.

B. Ad-Hoc Analysis of Story Boards

Without any methodological guidelines, the generalization of the above scenario's *could* start from Figure 3. From this scenario, one *could* identify that:

- first, the above two disks are moved to the middle tower (from State 2 to 4),
- then, the largest disk is moved to the right tower (in State 5),
- finally, the two other disks are moved to the right tower too (from State 6 to 8).

Moreover, one *could* observe that the act of moving from one tower to another tower is always supported by a third "auxiliary" tower: on Figure 3, moving the tallest disk (called "diskToMove") from the left to the right tower, is supported by the middle tower. Similarly, moving the smaller disk to the middle tower (states 2 to 4) is done by temporarily moving the smallest disk to the right tower. Within that scope, the "right" tower thus plays the role of "auxiliary" tower.

Moving back to Figure 2, one should remark that moving two towers is not always done with that "right" tower as "auxiliary" tower. Instead, when moving two disks from the left tower to the right tower, the smallest disk (called "upper" in Figure 2) is temporarily moved to the "middle" tower as auxiliary tower. To conclude, each "move" operation is defined by (1) the disk to move, (2) a target tower, and (3) an auxiliary tower.

Considering the Story Boards in more detail, one can identify that this move operation only affects the *lays_on* relation between different disks (and between the lower disk and the tower foot). Therefore, one can implement the move operation as a method, of class "HanoiAlgo", with three parameters: *diskToMove, to,* and *using*. The first argument is always of type *Disk* whereas the latter two can be either of type *Disk* or of type *TowerFoot*. Therefore, the type hierarchy shown on Figure 4 includes a shared superclass for *Disk* and *TowerFoot*.

Still, one can argue that the ad-hoc consideration of the use case scenario's included some non-trivial considerations that may have been facilitated better with some methodological guidelines. At least, a complete development methodology

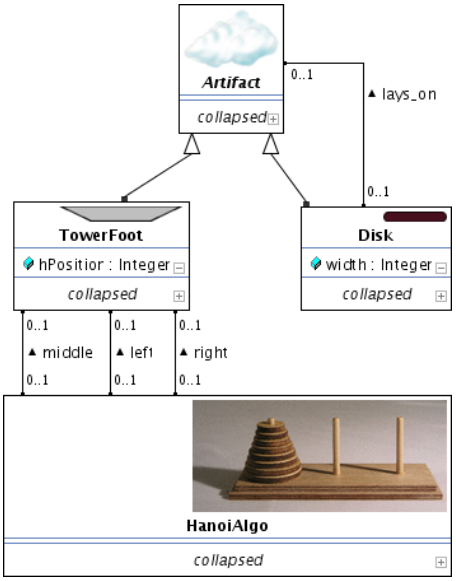


Figure 4. Domain model of the presented solution to the *Towers of Hanoi*.

should provide a means to document that an object (such as "middle") within a particular scenario plays a well-defined role (such as "auxiliary") in the context of an envisioned method execution. This motivates why Diethelm et al. are working on a set of rules and heuristics, some of which are applied in the next section.

C. Systematic Analysis of Story Boards

This section revisits the analysis of the three Story Boards in order to illustrate how (and to what extent) an implementation for the *move* operation can be derived systematically.

First of all, one should already make explicit that the three Story Boards are actually modeling the execution of the same method. More specifically, all three scenario's describe how disks should be moved from the left tower to the right tower, using the middle tower. Therefore, one should link the scenario's to one precise method by sending a *moveLtoR()* message to the *algorithm* object within the first state. Then, the *algorithm* object responds by sending a more generic *move(disk, to, using)* message to itself with the *diskToMove*, *right*, and *middle* nodes as arguments.

One can agree that moving to the consistent end states of the three scenario's is the responsability of that *move* method. In general, the derivation strategy prescribes that a Story Diagram is created for each method, where concrete object names are replaced by variable names. Since the actual work is done in the *move* method, the Story Diagram for the *moveLtoR* message can already be derived from the initial states of the Story Boards. As Figure 5 shows, this Story Diagram maps directly to the first state of the Story Boards. Operationally, the Story Diagram should be interpreted as follows: if there are three towers associated with the algorithm object under consideration, and the left tower holds a disk, then move that disk from the left tower to the right tower using the middle tower.

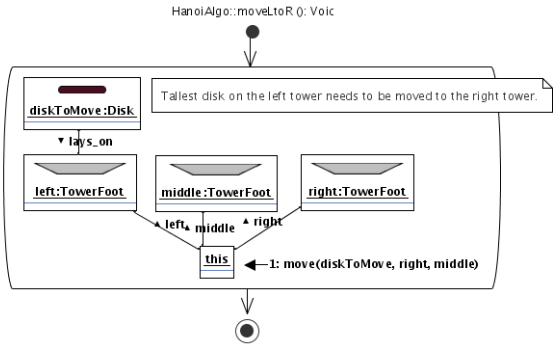
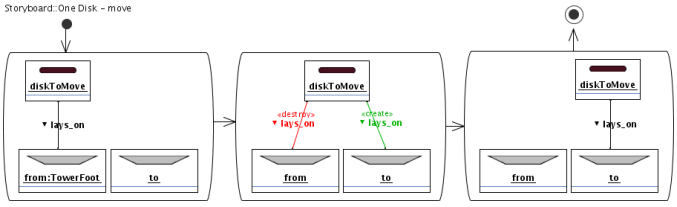
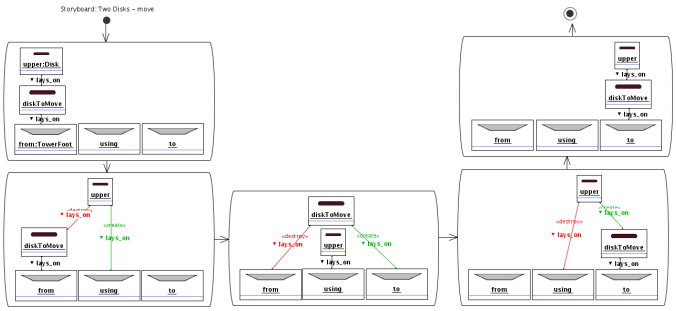


Figure 5. Story Diagram modeling that all disks should be moved from the left to the right tower.



(a) Scenario for 1 Disk.



(b) Scenario for 2 Disks.

Figure 6. Story Boards constraining the *move* method.

At this point, the new development challenge has become the definition of a general description of the *move* method. Therefore, all scenario's (Story Boards) are decorated in terms of that method. With the calls to the *move* operation properly added to the three Story Boards, one can substitute the names of the objects that are passed as parameters to derive the scenario's that constrain the *move* method. The Story Boards from Figures 1 and 2 are thus transformed into the Story Boards shown on Figure 6 (a) and (b) respectively.

The latter figures show that objects that are irrelevant for the execution of the newly created method are not shown on the extracted scenario: Figure 6 (b) does not include the *algorithm* object. If necessary, that object could still be accessed as the *this* object, but the *from*, *using* and *to* objects are accessed as method parameters instead of through a link navigation from the *Algorithm* instance. Also note that Figure 6 (a) also does not display the *using* tower since that object does not affect the scenario for moving one disk.

Thanks to the simplification of the Story Boards, the derivation of a Story Diagram is slightly simplified. The next systematic step in the derivation of Story Diagrams from Story

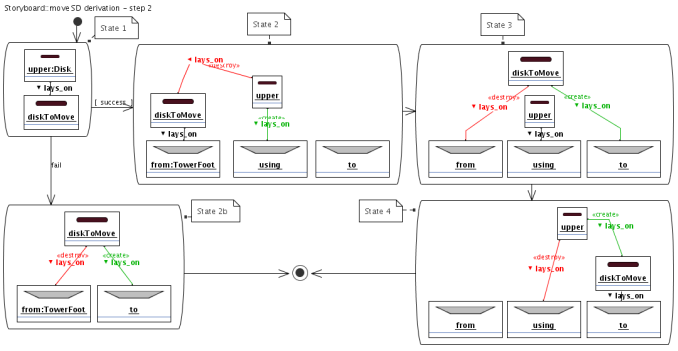


Figure 7. Deriving a Story Diagram from the Story Boards for one and two disks.

Boards consists of creating an "empty" Story Diagram for the *move* method. Such an empty Story Diagram consists of an initial state that only holds and initial state and an end state. In the next derivation step, the initial states of the Story Boards are added as next-states from the initial state. Since the two transitions leaving the initial state contain no guard condition, the Story Diagram is ambiguous.

This conflict is resolved by inspecting the differences between the two newly created states and considering their differences. Since the *using* object is not used in the state derived from the initial state of Figure 6 (b), the only difference between that state and the state derived from the initial state of Figure 6 (a), is that in the former state, the *diskToMove* is connected to an upper object by means of the *lays_on* association. Therefore, the conflict is resolved by inserting a state before these two states. The only purpose of this state is to check the presence of that upper object.

Figure 7 shows the Story Diagram at this stage of the derivation process. The first state tests whether or not the *move* method is executed in the context of the scenario represented by the Story Board from Figure 6 (b): it checks whether or not there lays a disk on top of the disk to move. If this pattern is found, subsequent states from Figure 6 (b)'s Story Board need to be executed: therefore, these states are inserted after the "success" transition that leaves the first state from Figure 7's Story Diagram. Otherwise, the states from Figure 6 (a)'s Story Board need to be executed. Therefore, these states are inserted after the "failure" transition that leaves the first state of Figure 7's Story Diagram.

After thus removing the ambiguities between the two scenario's under consideration, one should attempt to eliminate duplicate patterns. In the example given, one can observe that in states 2 to 4 from the current Story Diagram (shown on Figure 7), disks are moved using the behavior captured by state 2b. More specifically, since in state 2 no other disks lay on top of the *upper* disk, it is moved to the *using* tower by calling *move(upper, using, to)*. That call will be resolved by entering state 2b. Similarly, state 3 can be modeled as move(diskToMove, to, using) and state 4 can be modeled as *move(upper, diskToMove, from)*. The latter call would not be resolved correctly by entering state 2b recursively since the *to* object is not of type *TowerFoot* in this case: instead, *diskToMove* is of type Disk.

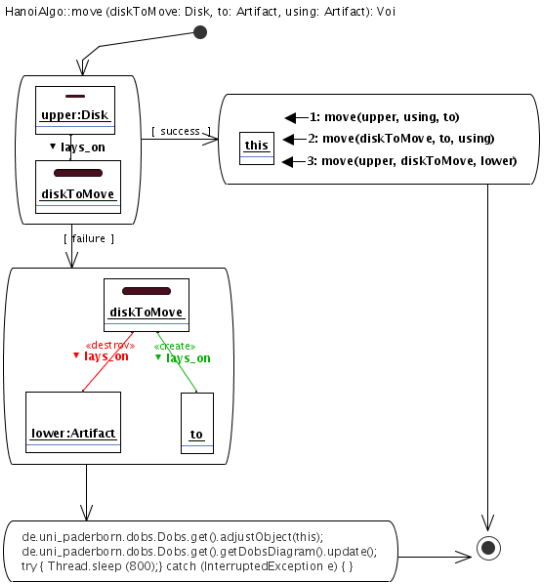


Figure 8. Story diagram for *move*, modeling how one disk can be moved in general.

Similarly, the *move(upper, using, to)* call would lead to problems in state 2b since the *from* object is representing the *diskToMove* object from the caller. A final typing conflict is detected when taking into account the scenario for moving three disks. More specifically, in the state equivalent to state 3 from the Story Board displayed on Figure 3, a *move(smallest, smaller, diskToMove)* call would pass the *diskToMove* object (of type *Disk*) as the parameter called *using* (and of type *TowerFoot*).

In summary, these typing conflicts require one to change the type of the *to* and *using* parameters of the *move* method, as well as the type of the *from* object within this method, from type *TowerFoot* to type *Artifact*.

Figure 8 presents the final version of the Story Diagram modeling the *move* method. This Story Diagram is consistent with all use case scenario's described by the above Story Boards. The state containing only Java code is related to interaction with the user interface of Fujaba's debugging/simulation environment. The resulting application can thus be used to animate the movement of an arbitrary number of disks across three towers.

D. Evaluation and Outlook

This section evaluates the presented application of Story Boarding and motivates why we are interested in extensions of the Story Diagram language.

The Towers of Hanoi example clearly illustrates some advantages of the Story Driven Modeling methodologies based on noun-verb analyses and CRC cards. First of all, Story Boards present a unique approach to represent use case scenario's explicitly. During analysis, such scenario's are the primary means of communication between domain experts and software engineers. They allow one to think in terms of concrete objects instead of forcing one to reason about more abstract classes directly. The Story Boards for moving one,

two and three disks across different towers are as concrete as their real-world counterparts.

This correspondence is naturally constrained by the quality of the underlying domain model. To illustrate that other Story Boards may have a wider mental gap to the real world, consider the Story Boards, Class Diagram and Story Diagrams from the original paper from Diethelm et al. [9]. In that paper, the underlying domain model does not include the concept of a tower. On the one hand, this simplifies all models since there is no need for the abstract *Artifact* concept which can be said to complicate Figure 8. On the other hand, Diethelm models the foot of a tower by means of a permanent *Disk* instance which can be said to be confusing.

Concerning the derivation of Story Diagrams from Story Boards, Section I-B illustrated the need for a systematic derivation strategy. Preliminary strategies have been discussed in the context of the development of a board game simulator [11], a library system [8] and a shuttle control system [6], [7]. Section I-C applied the most systematic procedure so far [7]. Although the derivation strategy systematically resolved the conflict between the recursive step of the *move* method and its base case, it did not guide the following steps:

- removing the duplication between state 2 and states 2b, 3 and 4 from Figure 7 by replacing the Story Patterns from states 2b, 3 and 4 by recursive calls to the *move* method.
- changing the types of the *lower*, *using* and *to* objects from *Disk* or *Towerfoot* to *Artifact*.

One can argue that the first action strongly resembles Diethelm's strategy "(d) identify similar activities within the method body and try to merge them". However, this strategy clearly needs to be refined since "merging" can be realized by means of an iterative loop as well as by means of recursive calls. Similarly, the strategy "(e) Resolve conflicts in the resulting control flow by adding appropriate branching conditions" should be extended to take into account that apart from control flow conflicts, the "merging" step may also introduce typing conflicts. Section I-C illustrated that the use of common superclasses may resolve such conflicts. However, these strategies, as well as the existing ones, need to be investigated in more detail to ensure the generality of the derivation process.

II. Story Diagrams

While a complete discussion of the Story Diagram syntax can be found in the work of Zündorf [2], this section presents some key concepts of the language that should make it accessible for readers familiar with mainstream object-oriented programming languages such as Java. More specifically, it relates the concrete Story Diagram syntax to more general graph transformation concepts and to Java code that realizes the graph transformation concepts. Subsection II-A and II-B introduce the reader to primitive graph rewriting rules while subsection II-C covers more complex control flow constructs.

A. Graph Transformation Basics: Left- and Right-hand Sides

The most primitive and probably most widely known concepts of graph transformation systems are a *rule* (or *production*), a *match* (or *occurrence*) and a *transformation step* (or *direct derivation*). First of all, a graph transformation rule consists of a graph pattern, that needs to be looked up in a host graph. This pattern is often called the left-hand side of the rule because a large number of graph transformation languages visualize this pattern on the left part of the rule. Secondly, a rule contains a graph pattern that should be present in the host graph after the first pattern has been found. For similar reasons, this pattern is often called the right-hand side of the rule.

The precise semantics for matching the pattern on the left-hand side has been formalized in terms of category theoretical concepts. More specifically, one has defined what kind of morphisms need to exist between the nodes and edges in the left-hand side of a rule and the nodes of a match in a host graph. Several variants have been proposed, of which major differences consist of:

1) whether or not different nodes in the left-hand side are allowed to be mapped to one node in the host graph, and
2) whether or not a node can be deleted when such a step would produce edges related to only one node (i.e., *dangling edges*).

The first difference relates to the injectivity of matches. In MoTMoT [12], matches are *non-injective* by default but an additional "application condition" can be used to ensure a particular rule only matches injectively. The official Fujaba version of Story Diagrams supports injective matching by default [10].

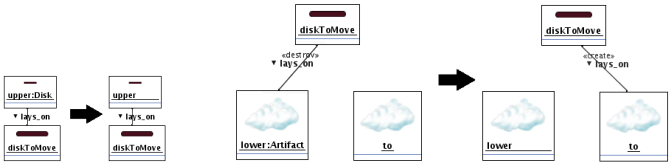
The second difference distinguishes the "double pushout approach" [13], that statically enforces the explicit "gluing condition" to ensure all node and edge morphisms are homomorphic, from the "single pushout approach" [14], that automatically deletes dangling edges when they happen to be produced at rule execution time. This thesis relies on the single pushout semantics because it offers the most freedom to the transformation modeler.

The following algorithmic steps, based on [15], clarify simplistically how a graph transformation rule can be applied to a host graph G_{host}:

1) Identify the left-hand side G_{LHS} within the host graph G_{host}. Note that he fulfillment of specific *application conditions* must be checked. These conditions can be *negative*, in which case a pattern will only be matched if the specified structure is not present in the host graph.
2) Delete from G_{host}, each element (i.e., node or edge) e_{lHost}, that corresponds to an element e_{lRule} from G_{LHS} when e_{lRule} does not occur in the right-hand side G_{RHS}.
3) Create a graph element e_{rHost} in G_{host} for each element e_{rRule} that is part of G_{RHS} but is no part of the left-hand side G_{LHS}.
4) For each node n_{rRule} in G_{RHS} that carries "attribute value assignments", update the related attribute values of the node n_{rHost} in G_{host} that corresponds to n_{rRule}.

B. Concrete Syntax: Merging LHS and RHS

Several graph transformation languages (such as Progres [16]) display the left- and right-hand sides separately. How-



(a) (b)

Figure 9. Alternative syntax for primitive graph rewriting rules.

ever, this has some disadvantages.

First of all, the transformation writer needs to label all elements from both sides to indicate what elements occur on both sides. Secondly, when reading a transformation rule, one needs to investigate these labels carefully to find out what nodes are part of the left-hand side without occurring on the right-hand side and vice versa. Finally, in order to compensate the previous disadvantage, one needs to maintain the same layout for all elements that occur both on the left- and right-hand side of the rule.

Therefore, other languages, and Story Diagrams in specific, eliminate the duplicate specification of elements occurring on both sides of the rule such that the differences between the two sides becomes more visible. Nodes and edges marked with the $\ll destroy \gg$ stereotype appear only on the left-hand side of the rule. As indicated by the name and following the original semantics, such elements are deleted. The stereotype $\ll create \gg$ marks elements only used on the right-hand side. Clearly, such elements need to be created.

Following these syntactical conventions, the rewrite rules from Figure 8 can also be visualized as on Figure 9. For these small rewrite rules, the drawbacks discussed above may be neglectable.

C. Control Structures

While the need for control structures may be evident for general purpose programming and while the *Hanoi* example from Section I-A already illustrated that Story Diagrams support explicit control flow constructs, it should be noted that there are other graph transformation languages that do no support conditionals, loops or invocations. Therefore, this section first motivates why such control structures are useful in a model transformation setting and then presents the Story Diagram control flow constructs.

1) Why: We will use a transformation that has been discussed in [17] to illustrate that a pure graph grammatical can lead to poorly readable transformation models. The poor readability is caused by the need for artificial node attribute assignments that could have been modeled more explicitly by means of control structures. The example concerns the in-place transformation of a Petri-Net model. The transformation realizes the operational semantics of *Place Transition* nets. The metamodel, transformation and sample models have been realized using $AToM^3$ and can be obtained from the author.

In general, Petri-Nets consist of *Places*, *Transitions* and *arcs* between these concepts. A Place can hold a finite amount (its *capacity*) of tokens. The set of Places that can be reached by the arcs that leave a particular Transition are called that Transition's *ouput places* while the Places that have an outgoing arc to a Transition are called that Transition's *input places*. Each arc carries a natural number attribute called *weight*, that defaults to 1. The operational semantics of Place Transition nets prescribes when a transition can be *fired*. When a transition fires, a well-defined amount of tokens is taken from all input places while another amount of tokens is added to that Transition's output places. The individual amounts are based on the weights of the respective arcs. A Transition t can be fired when:

- all input places of t contain at least as much tokens as indicated by weight of the arc that leads to t,
- all output places of t can be incremented with the amount of tokens indicated by the arc that leaves t, without exceeding their capacities.

The primitive rules of the graph transformation that realizes this semantics take care of a small modification only: one rewrite rule removes tokens from an input place, another rule adds tokens to an output place, and yet another rule checks whether a transition can be fired.

One of the challenges that needs to be resolved is that as soon as tokens have been removed from one input place, the transition that has been fired might no longer satisfy the conditions for being fired. Therefore, when a transition is fired, it needs to be flagged with a particular "state" attribute that will only be reset after all output places have been incremented.

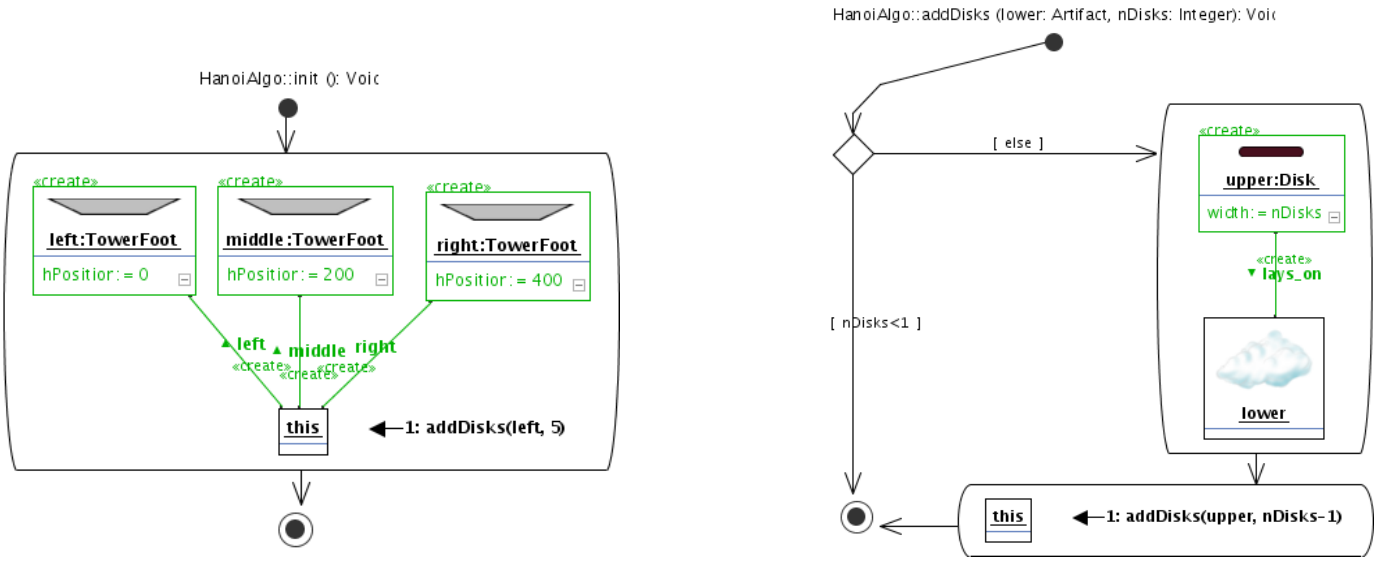
Moreover, it would be illegal for the transformation system to fire two neighbouring transitions simultaneously. Suppose for example, that two transitions have exactly one input place that happens to be the same. Moreover, suppose this place holds exactly one token. Semantically, only one of the two transitions is allowed to be fired. However, without additional mechanisms, a rule engine could first execute the rules for checking firability on both transitions and execute the rules for transferring tokens across both transitions afterwards. Again, this situation can be prevented by setting a transformation-wide property such that the firing of transitions is synchronized.

These two examples illustrate that conditional and looping behavior can be realized by means of node attributes. However, these attributes are purely technical and would pollute the underlying metamodel. Moreover, the control flow is very implicit while this thesis aims to model transformations in a human-friendly manner. Another clear example of an undesirably explicitly defined control flow can be found in [18], where the flow between grammar rules is driven by technical node annotations.

2) What: Story Diagrams offer five concepts for modeling a transformation's control flow explicitly: (i) sequential composition, (ii) conditionals, (iii) while loops, (iv) iterative loops, (v) method calls.

Sequential Composition enables one to express that a story pattern b should be evaluated one step after another story diagram a, independently of whether a's left-hand side could be matched. This is specified by a guard-less transition between the states of a and b.

The Story Diagram from Figure 10 (b) illustrates the use of an explicit conditional state.



(a) Creating the three towers.

(b) Adding disks to a tower.

Figure 10. Story diagrams for initializing the data structure for the *Hanoi* example.

Such a state is not related to any rewrite rule and does not have any other constraints or side-effects associated with it. However, it may put additional focus on the conditions of its outgoing transitions. The example illustrates that Story Diagrams support more condition types that the *success* and *failure* conditions introduced in the context of Figure 8's Story Diagram. Note that the latter two condition types only apply when a state does have a pattern associated with it.

A while loop consists of a story pattern "conditional", a *success* transition to an arbitrary sequence of "body" states, a sequential step back to "conditional", and a *failure* transition to the end of the loop. Note that the "conditional" story pattern may have side-effects. Such side-effects are even mandatory to terminate a loop with an empty sequence of "body" states. Otherwise, the sequential transition from "conditional" to itself would be fired endlessly as soon as the left-hand side of "conditional" would match.

A frequently required task is walking iteratively over a set of pattern matches, or simply "handling all matches of a patricular pattern, once". When realizing such a traversal using a while loop, one needs to mark elements from the matches in such a manner that they will not match another time. Instead of this low-level approach, one may want to use a more explicit language concept, called "iterative loops". Such loops visit the matches from their "conditional" pattern only once.

The final control flow construct is the method call. For object-oriented modelers, the concept of such a call should be more than familiar. Since each Story Diagram maps directly to a method, for which a Java (or C++, ...) implementation can be derived automatically, a call to a Story Diagram can be realized by means of a Java method call. Thus, the Java virtual machine's infrastructure for stack frames is reused. Still, some may argue that a more understandable formalization of Story Diagram method calls would not rely on the infrastructure of other languages.

Therefore, Zündorf illustrated how three simple concepts "H_Stack", "H_Frame" and "H_Variable" enable one to model the behavior of a method call explicitly, using more primitive Story Diagrams [2].

Conceptually, the use of control flow structures simplifies the subgraph matching problem that is needed within a graph transformation engine, significantly. Engines no longer need to search for morphisms from the left-hand side graphs across the complete host graph. Instead, the matching algorithm builds incrementally upon results from previously matched rules. Notationally, Story Diagrams provide the so-called "bound" property to indicate what nodes are already found within the host graph. Only other nodes from the host graph need to be looked up by the matching engine.

Bound nodes come from two sources: first of all, object attributes and method parameters are available to all patterns in a Story Diagram. This is illustrated by the Story Diagrams from Figure 10 (a) and (b): after creating the node *left* in the first Story Diagram, it is passed to the second Story Diagram where it is represented by the bound node called *lower*. A second source of bound nodes is the following: all nodes that have been matched or created by one rule become implicitly available to all other rules that are scheduled sequentially later, and in the same or a deeper scope. Again, Zündorf has illustrated how this implicit information can be made explicit by relying on the "H_Stack", "H_Frame" and "H_Variable" concepts.

When supporting method calls, one requires a means to return an object or a value to the caller. In Story Diagrams, this is supported by means of end states that have the name of a particular bound node or that have a specific value as their name. When a transformation reaches such an end state, the related node or primitive value is returned to the caller.

III. Discussion and Conclusions

This paper introduced the reader to the Story Driven Modeling process, by discussing the *Towers of Hanoi* case study.

In summary, this example illustrates that the Story Driven Modeling methodology provides a promising basis for the systematic derivation of complete behavioral models. One should remark that the two heuristics for removing ambiguities and duplications are generic in the sense that they are independent of the concrete behavioral target language (that is, of Story Diagrams in this case). Nevertheless, the notion of a concrete ambiguity does depend upon the behavioral modeling language in use. More specifically, one could technically extend Story Diagrams with an operator for moving elements according to the requirements of the *Hanoi* problem.

With such an operator in place, the Story Boards from Figure 6 (a) and Figure 6 (b) would not be in conflict. Instead, they could be generalized into one Story Diagram that would apply the *Hanoi*-specific language construct, thus covering the behavior of all *Hanoi* scenario's. Of course, the definition of a Hanoi-specific construct may not make much sense since it would not be generally applicable for solving a wide range of application (or model transformation) problems.

Therefore, the main focus of our research lies on evaluating the applicability of Story Diagrams on the one hand while extending it for model transformation purposes when appropriate [19]. Extensions should solve problems from concrete case studies and should be applicable to more than one transformation type. This motivates the need for a good taxonomy of the domain of model transformations [20].

Obviously, the Story Driven Modeling process does not provide the *Silver Bullet* [21]: first of all, the quality of Story Boards depends on the concepts that are used for articulating the use case scenarios. The selection of such concepts is a human task that cannot be manipulated by a modeling language. Secondly, the method to generalize Story Boards into Story Diagrams is subject to ongoing research. Consequently, the development of Story Diagrams does require abstraction capabilities from the modeler.

However, when targeting the model transformation community, developers are at least educated software engineers or experienced programmers. Such developers may find Story Diagrams a promising formalism: on the one hand, the underlying object-oriented paradigm ensures the language concepts are familiar to the large audience of *Java*, *C#* and *C++* developers. On the other hand, rewrite rules are supported as first-class language constructs. Moreover, several control flow constructs enable one to model dependencies between such rules explicitly. Section II-C motivated why such constructs are essential for modeling transformations in a human friendly manner.

References

[1] Albert Zündorf. From use cases to code—rigorous software development with uml. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 324–325, New York, NY, USA, 2001. ACM Press.

[2] Albert Zündorf. Rigorous Object Oriented Software Development, 2002. Habilitation Thesis, University of Paderborn. Draft version 0.3.

[3] Russell J. Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[4] T. Halpin. Augmenting UML with fact orientation. In *HICSS'01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 3*, Washington, DC, USA, 2001. IEEE Computer Society.

[5] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM Press.

[6] Ira Diethelm, Leif Geiger, Thomas Maier, and Albart Zundorf. Turning collaboration diagram strips into storycharts. In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'02*, Orlando, Florida, USA, 2002.

[7] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic story driven modeling, a case study (Paderborn shuttle system). In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE'04*, Edinburgh, Scottland, 2004.

[8] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic story driven modeling, a case study (library system). Technical report, University of Kassel, 2004.

[9] Ira Diethelm, Leif Geiger, and Albert Zündorf. UML im unterricht: Systematische objektorientierte problemlösung mit hilfe von szenarien am beispiel der türme von Hanoi. In *Erster Workshop der GI-Fachgruppe Didaktik der Informatik*, Bommerholz, Germany, 10 October 2002.

[10] Holger Giese, Albert Zündorf, Andy Schürr, Bernhard Westfechtel, and Leif Geiger. Fujaba toolsuite. In *International Fujaba Days*, 2003–2007.

[11] Ira Diethelm, Leif Geiger, and Albert Zündorf. Teaching modeling with objects first. In *WCCE 2005, 8th World Conference on Computers in Education*, Cape Town, South Africa, 2005.

[12] Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). motmot.sf.net, 2007.

[13] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, and Anika Wagner. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter Algebraic Approaches to Graph Transformation - Part I: Single Pushout Approach and Comparison with Double Pushout Approach, pages 163–245. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[14] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach, pages 247–312. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[15] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer-Verlag, 2005. Edited by Sami Beydeda and Volker Gruhn.

[16] Andy Schürr. Progres: A visual language and environment for programming with graph rewrite systems. Technical Report AIB 94-11, RWTH Aachen, Fachgruppe Informatik, 1994.

[17] Pieter Van Gorp, Hans Schippers, Serge Demeyer, and Dirk Janssens. Students can get excited about formal methods: a model-driven course on petri-nets, metamodels and graph grammars. In Miroslaw Staron, editor, *The 3rd Educators' Symposium of the 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Symposium Proceedings*, Göteborg, Sweden, 2007. IT University of Göteberg, Department of Applied Information Technology.

[18] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, Genova, Italy, 2006. Springer Verlag.

[19] Pieter Van Gorp, Hans Schippers, and Dirk Janssens. Copying Subgraphs within Model Repositories. In Roberto Bruni and Dániel Varró, editors, *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, pages 127–139, Vienna, Austria, 1 April 2006. Elsevier.

[20] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 03 2006.

[21] Jr. Frederic P. Brooks. *The Mythical Man Month*. Addison–Wesley, anniversary edition, 1995. ISBN: 0–201–83595–9.