Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams

Bart Meyers
University of Antwerp
Antwerpen, Belgium
bart.meyers@student.ua.ac.be

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

ABSTRACT

Transformation rules can be controlled explicitly using language constructs such as a loop or a conditional. This approach is realized in Fujaba's Story Diagrams, in VMTS, MOLA and Progres. Alternatively, transformation rules can be controlled implicitly using a fixed strategy. This approach is realized in AGG and AToM^3. When modeling transformation systems using one approach exclusively, particular aspects could have been expressed more intuitively using the other approach. Unfortunately, most (if not all) transformation languages do not enable one to model the control of some rules explicitly while leaving the control of other rules unspecified. Therefore, this paper proposes the extension of Story Diagrams with support for implicit rule scheduling. By relying on a UML profile and on higher order transformations, the language construct is not only executable on the MoTMoT tool, but on any tool that supports the standard UML syntax for Fujaba's Story Diagrams.

Keywords

Transformation Languages, Rule Scheduling, Higher Order Transformations, Language Engineering

1. INTRODUCTION

Transformations are critical in model-driven development. Since homemade modeling languages may be defined and integrated at any time in the development process of a software system, transformations have to be tailored accordingly to integrate these new languages before they can actually be used. Therefore, a highly expressive transformation language is very useful, because it facilitates defining the needed transformations.

When defining transformation languages, certain choices are made. The language can be imperative (i.e., operational) or declarative [8, Chapter 2]. Explicit rule scheduling mechanisms (e.g., conditionals) tend to be called *imperative* since they enable one to model the execution of transformation rules in terms of the *state* of the transformation system. Languages with implicit rule scheduling (such as AGG) tend to be called *declarative* due to the absence of an explicit state concept. There is no better choice in this matter. For certain problems implicit rule scheduling feels more intuitive, sometimes explicit rule scheduling turns out to be convenient. Therefore, this paper introduces a language construct for the integration of implicit rule scheduling in an imperative language. The integrated language is *hybrid* (imperative as well as declarative) with regards to rule scheduling.

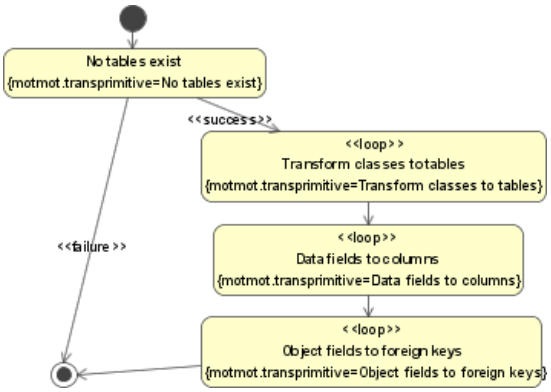

Figure 1: Control flow of a transformation from class diagrams to a relational database schema.

The remainder of this paper consists of the following: Section 2 motivates the need for the hybrid transformation language by means of an example. Section 3 describes a higher order transformation for mapping hybrid transformation models to fully imperative transformation models in ordinary Story Diagrams. This defines a compiler for the new language construct. Section 4 presents related work and Section 5 discusses future work. Finally, Section 6 summarizes with a conclusion.

2. IMPLICIT RULE SCHEDULING

Consider the example of Figure 1, which represents a simplified model transformation from class diagrams to relational database schemata. This example has been used for comparing transformation languages before [1]. Figure 1 displays a story diagram that applies explicit rule scheduling exclusively. Throughout this paper, the UML profile for story diagrams is used. The benefits of using a profile rather than metamodels are discussed in Schippers et al. [6].

In the UML profile, rules (annotated class diagrams) are embedded in a control flow (annotated activity diagrams) by using a tag named *transprimitive* whose value points to the UML package containing the transformation rule (i.e., classes and associations annotated with $\ll create \gg$, $\ll destroy \gg$, etc. [8]). In effect, the value of the *transprimitive* tag states which rewrite rule needs to be executed when the transformation system is in a particular state. Fujaba realizes the same semantics but differs syntactically by visually embedding the rewrite rules instead of referring to their name.

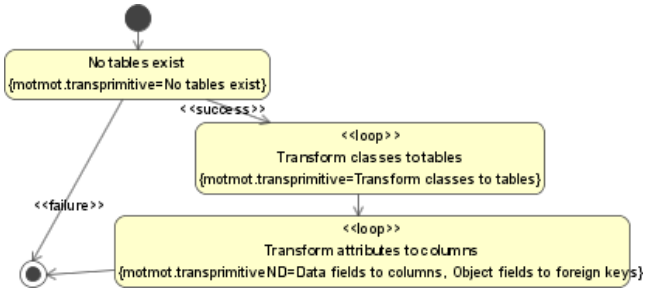

Figure 2: Example of the usage of the new language construct.

In Figure 1, the first rule that has to be executed checks whether there are tables in the database. If some tables exist in the database, the algorithm ends since otherwise existing data may be corrupted by the simple transformation. If not, the actual creation of schema elements is started: first, classes are transformed to tables. Next, class attributes must be transformed. There are two types of attributes: object references and primitive values. It turns out that the transformation of the two kinds of attributes can be modeled elegantly with two transformation rules. Attributes of a simple data type `t` become columns of type `t`. Object attributes of class `T` become columns of type integer containing key references to the primary unique ID column of table `T`. Additionally, a foreign key constraint is added to the database. Therefore, in Figure 1, all data fields are transformed, and then all object fields are transformed.

Having to express the transformation of attributes in two sequentially executed rules decreases the quality of the transformation model in several ways. First, one has to impose an order on these two rules, which is useless and has no meaning. This is a clear case of over-specification. Secondly, the transformation of all attributes is conceptually one action, and should be modeled as such.

Alternatively, one could have modeled the transformation using implicit rule scheduling. However, in that case, the "no tables exist" test (this kind of sanity checks are rather common at the start of model transformations) could not have been scheduled before the other rules without having to rely on hand-written code or other tool-specific approaches. This is a reason why modeling in a language using explicit rule scheduling is a good choice for this problem.

2.1 A new language construct

It turns out that both implicit and explicit rule scheduling are needed to model the example of Figure 1 in a decent way. Therefore, this paper introduces a new language construct for story diagrams that allows implicit rule scheduling. Consider Figure 2 as an example of the usage of this new construct. Analogue to the *transprimitive* tag definition, a new UML tag definition *transprimitiveND* is proposed that can be used for the state that transforms attributes. However, a *transprimitiveND* state can reference more than one UML Package and chooses non-deterministically in which order the packages are executed, hence "ND" in the name.

More general, consider a set of rules that can be executed in

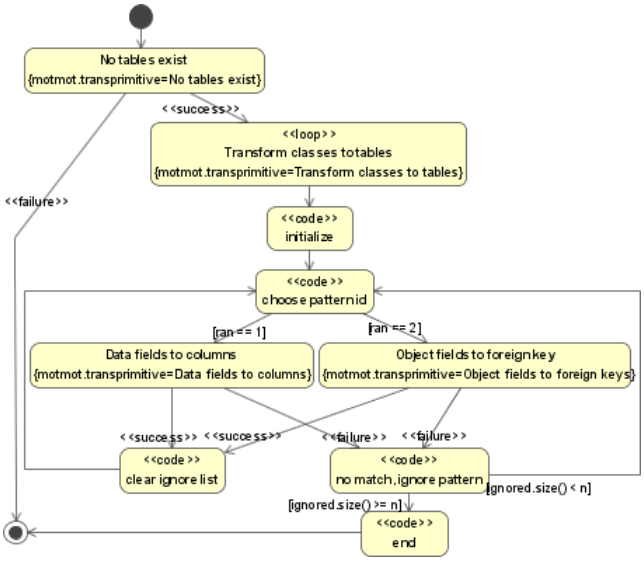

Figure 3: Fully imperative equivalent of Figure 2.

any order. Such rules need to be executed until all of them fail to match. Every time a rule of this set is evaluated, it is executed, and all the other rules of the set have to be checked again in the next iteration, because applying a rule to a model can change the model.

2.2 The fully imperative equivalent

The new construct must be transformed to an equivalent which is solely written in plain story diagrams. Otherwise, our contribution would probably only become supported by our own tool. Such an imperative equivalent for the example from Figure 2 is shown in Figure 3. After the `Transform classes to tables` state, the `initialize` state is entered, where some variables that will be used are declared. The ≪*code*≫ stereotype denotes that the state corresponds to (Java) code instead of a transformation rule. More in detail, an integer `n` is set to 2, as there are two rules (`Data fields to columns` and `Object fields to foreign keys`) that can be executed. A list `ignored` is initialized, which will represent the rules that must be ignored because they didn't match in the current model state.

After the `initialize` state, a rule is chosen non-deterministically in the `choose pattern id` state, also a code state. In fact, a random number generator produces an integer ranging from 1 to `n`, which represents the randomly chosen rule. Moreover, this generated integer must not be contained in the `ignored` list. According to this random number, one of the *transprimitive* states which represent the actual transformation rules is entered.

If the rule matches, the ≪*success*≫ transition is followed. This transition leads to a state that ensures that `ignored` is cleared, as the ignored rules must be evaluated again because the state of the model might have changed. Then, a new rule can be chosen in the `choose pattern id` state for execution.

If the rule didn't match, it would be useless that this rule would be coincidentally chosen in the next iteration, so the

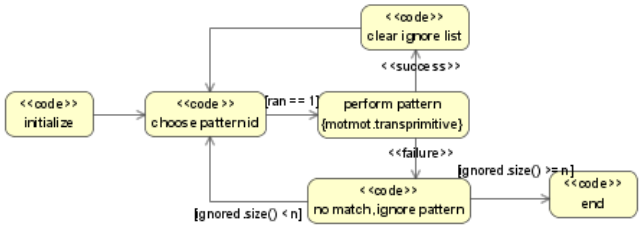

Figure 4: Prototype that is put into the input model.

rule is added to the `ignored` list. As long as there are rules available for execution, a new rule can be chosen for the next iteration. If there are no rules left for execution, i.e. they are all in `ignored`, the algorithm ends. The `end` state does nothing, but is added in order to end the equivalent with a transition without a guarding expression.

This results in an equivalent for unconstrained implicit rule scheduling, which schedules all the rules with equal priority. This algorithm can easily be extended to support constrained rule scheduling, like the use of priorities [2] (as realized in AToM^3) or layers [5] (as realized in AGG).

3. HIGHER ORDER TRANSFORMATION

In this section, a higher order transformation is proposed that transforms applications of the introduced new language construct to plain story diagrams.

The higher order transformation consists of a main loop that sequentially transforms each *transprimitiveND* state in the input model. Such states are transformed to their imperative equivalents as in Figure 3. These equivalents will be very similar to one another. Therefore, the common part is bundled into a separate generic prototype model, given in Figure 4. Starting from this prototype will avoid verbose rewrite rules in abstract syntax form. To use the prototype in the input model, it has to be moved to there. Roughly speaking, this is done in the following steps: first, the prototype model file is read. Secondly, all the states of the prototype model are moved to the input model. Thirdly, all the transitions of the prototype model are moved to the input model. Fourth, the target of some transitions is reassigned. And fifth, the source of some transitions is reassigned.

Interestingly, the second, third, fourth and fifth step can be executed independently. So in theory, a *transprimitiveND* state could be used to control the rewrite rules for these steps. Doing so would emphasize the similarity between these steps, thus improving the structure and readability of the overall transformation model. However, we have not yet *bootstrapped* the higher order transformation [4]. Instead, it is modeled using plain story diagrams (i.e., the language without the *transprimitiveND* construct).

Once a prototype is created in the input model, the (Java) code in the `initialize` code state must be changed in order to initialize `n` with the actual number of rules. Then, for each rule of the *transprimitiveND* state, a new `perform pattern` state has to be created, with its according transitions. Figure 5 represents a slightly simplified version of the pattern in the higher order transformation that performs this task. A new state `performState` and transitions from `choose pattern id` and to `clear ignore list` and `no match, ignore pattern` are created.

When each rule of the *transprimitiveND* state is added in a *transprimitive* state, a few elements, including the original *transprimitiveND* state must be removed from the input model. Once we bootstrap the higher order transformation, we will model this behavior using a *transprimitiveND* state with one rewrite rule for each element that needs to be removed. When all *transprimitiveND* states are transformed, the transformation completes.

4. RELATED WORK

The imperative realization from Section 2.2 executes rules sequentially, in a random order. Alternatively, one could execute the rules in parallel. It seems intuitive to rely on UML *Fork* and *Join* elements to model the parallel nature of a transformation system explicitly. However, neither version of Fujaba nor MoTMoT generates any code aimed at parallel execution (thread creation, synchronization, ...). Therefore, a higher order transformation approach such as the one presented in Section 3 does not seem to be applicable. Instead, one would have to extend the core of a particular story diagram tool. On the other hand, the use of standard UML elements (see Section 2.1) does apply. Instead of relying on UML activity diagrams as a standard language for controlling the application of rewrite rules, Syriani and Vangheluwe rely on DEVS [7].

The non-determinism of implicit rule scheduling can lead to unexpected results: in many cases, one rule for example creates elements that are used by another one and deleted by yet another rule. Since such dependencies can be introduced accidentally, dedicated analysis support is desirable in transformation tools that support languages with implicit rule scheduling. For example, the AGG tool offers a so-called Critical Pair Analysis (CPA [3]). To be applicable on the proposed hybrid language, CPA algorithms need to take into account nodes that are already bound from previously executed rules in a control flow.

5. FUTURE WORK

As an easy extension to this paper, the algorithm can be extended to support priorities or layers, as briefly stated in Section 2.2. Support for layers can be realized in another higher order transformation by putting together all rules of the same priority in *transprimiveND* states, then ordering them according to their priorities or layers. The higher order transformation from this paper can then transform the resulting model to a plain story diagram. In the case of priorities, the execution engine needs to re-evaluate all rules upon each iteration again, starting from rules with the lowest priority. Although this can be realized again using a higher order transformation, the one presented in this paper produces quite a different control structure.

The current realization aims at true nondeterminism by using random numbers. In some cases however, the order of the rules is irrelevant. More specifically, one may not care if the same order is used at all times. In these cases, the random number generator is nothing but a performance bottleneck. Therefore, we may extend our approach with another,

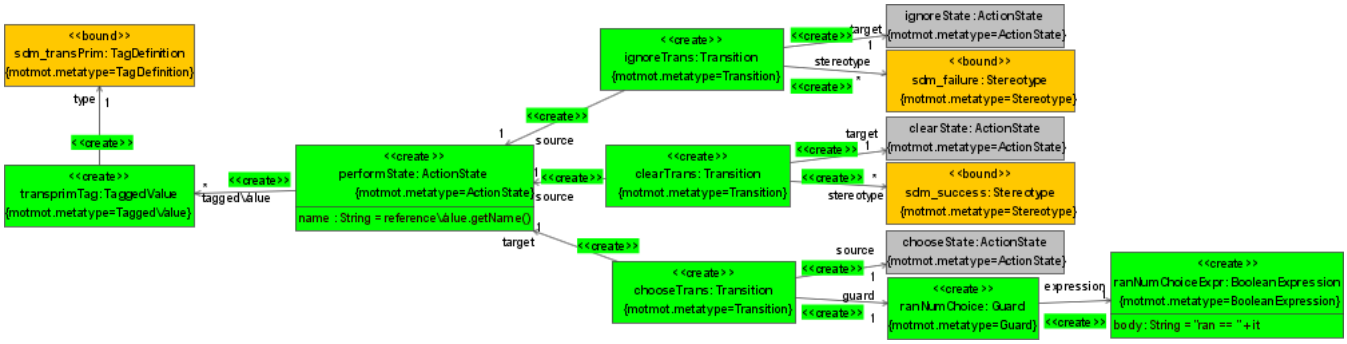

Figure 5: The rule in the higher order transformation that adds a perform pattern state and its transitions.

much simpler higher order transformation that imposes a particular order on the rules instead of guaranteeing randomness.

This paper discusses an algorithm for a *transprimitiveND* state in combination with a ≪*loop*≫ stereotype. Without a ≪*loop*≫, the *transprimitiveND* state ensures that all the rules are executed at most one time. A ≪*success*≫ transition will be followed if all rules did match once, and a ≪*failure*≫ transition will be followed if any of the rules did't match. As an example, suppose that in Figure 2, besides the `No tables exist` state, many other sanity checks have to be passed before the actual transformation can be done. All these checks could be referenced in one *transprimitiveND* state, avoiding a cascading effect of states with ≪*success*≫ and ≪*failure*≫ transitions, which could easily become very verbose and confusing.

This paper presents a transformation language that is hybrid with regards to rule scheduling. It is our ongoing work to realize a language that is hybrid with regards to other concerns, like *execution direction* and *change propagation* [8], too.

6. CONCLUSION

This paper discusses a standard syntax, the informal semantics and working tool support for a new language construct that allows users to use implicit rule scheduling in story diagrams. We illustrated its relevance and meaning by means of a toy example. As a more realistic example, we indicated where the language construct could even improve the readability and conciseness of its own compiler (i.e., that of its supportive higher order transformation). As a generic technique, this paper illustrated how profiles and higher order transformations enable language engineers to contribute new language contructs to a variety of tools (any version of Fujaba, MoTMoT, ...) without writing code specific to the editor or code generator of a particular tool.

7. REFERENCES

[1] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 120–127, 2005.

[2] S. M. Kaplan and S. K. Goering. Priority controlled incremental attribute evaluation in attributed graph grammars. In J. Díaz and F. Orejas, editors, *TAPSOFT, Vol.1*, volume 351 of *Lecture Notes in Computer Science*, pages 306–336. Springer, 1989.

[3] L. Lambers, H. Ehrig, and F. Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electron. Notes Theor. Comput. Sci.*, 211:17–26, 2008.

[4] O. Lecarme, M. Pellissier, and M.-C. Thomas. Computer-aided production of language implementation systems: A review and classification. In *Software: Practice and Experience*, volume 12, pages 785–824, 1982.

[5] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8:27–55, 1997.

[6] H. Schippers, P. Van Gorp, and D. Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2004. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation.

[7] E. Syriani and H. Vangheluwe. Programmed graph rewriting with DEVS. In M. Nagl and A. Schürr, editors, *International Conference on Applications of Graph Transformations with Industrial Relevance*, Lecture Notes in Computer Science, Kassel, 2007. Springer.

[8] P. Van Gorp. *Model-driven Development of Model Transformations*. PhD thesis, University of Antwerp, 2008.