Model checking AToMPM transformation systems with Groove

Jeroen De Busser

University of Antwerp

jeroen.debusser@student.uantwerpen.be

Abstract

We propose a set of model transformations in AToMPM that transform any AToMPM transformation system(abstract syntax, model instances, MoTiF transformation rules and rule scheduling) to its equivalent representation in Groove. This enables the use of Groove's model checking and state space generation tool-set.

Keywords:
AToMPM, MoTiF, Groove, Model checking, Graph Transformation Systems

1. Introduction

AToMPM is a tool for multi-paradigm modelling, and excels at creating visual domain-specific language modelling environments. Specifying the operational semantics of such a language can be done with graph transformation systems(GTS for short) in the MoTiF language, which is also modelled in AToMPM. It is however not built for analysis of these semantics, and thus checking if certain unwanted conditions cannot occur during simulation is a hard if not impossible task. Groove(Ghamarian et al., 2012) is a tool that specializes in generating labeled transition systems(LTS for short) of GTS's and performing model checking on them. This paper is structured as follows: Section 2 compares both tools' representations of (meta)models, transformation rules and rule scheduling. Section 3 details our solution and Section 4 gives an overview of possible future work. Section 5 concludes.

2. Comparison of AToMPM and Groove functionality

In the following subsections we'll go over each part of a model and its transformation system and look at the corresponding representations in AToMPM and Groove. We'll use a simplified version of an RPGame formalism as our example.

2.1. Meta-modelling for visual language engineering

Both tools have a way of describing the language the instance models are defined in. The meta-modelling capabilities of AToMPM, being a meta-modelling tool, are more elaborate by a large extent, but are mostly equivalent with those of Groove for our purposes(visual languages).

2.1.1. AToMPM

Modelling a visual language in AToMPM is done through a combination of the specification of the abstract syntax in the Class Diagram formalism and a concrete syntax in the ConcreteSyntax formalism. The latter makes it possible to have a truly visual syntax by defining the look of every class and association defined in the abstract syntax. Cardinalities can be defined and will be checked when adding a new edge while constructing an instance model. Arbitrary constraints can also be expressed in javascript code. An example metamodel for RPGame is given in Fig.1.

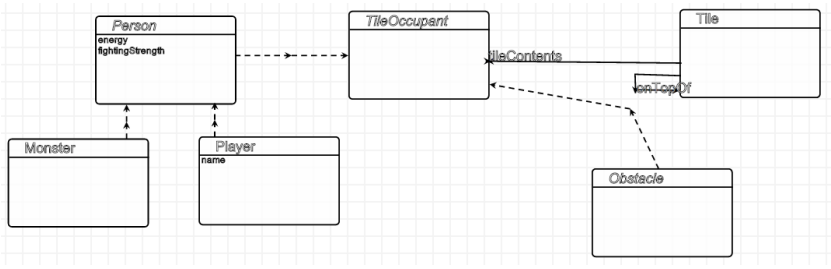

Figure 1: AToMPM metamodel

2.1.2. Groove

Groove has a notion of type graphs, which define what types of nodes are allowed and what edges can exist between them. It is roughly equivalent with a class diagram, except that the attributes of nodes can't be arbitrary types(the only types defined in Groove are integers, reals, booleans

and strings). It is possible to define cardinalities on these edges, and it is impossible to save any model with invalid ones. The only other constraint that can be checked while constructing a model instance in Groove is the occurrence of cyclical containment edges. The equivalent Groove type graph for the RPGame meta-model is given in Fig.2.

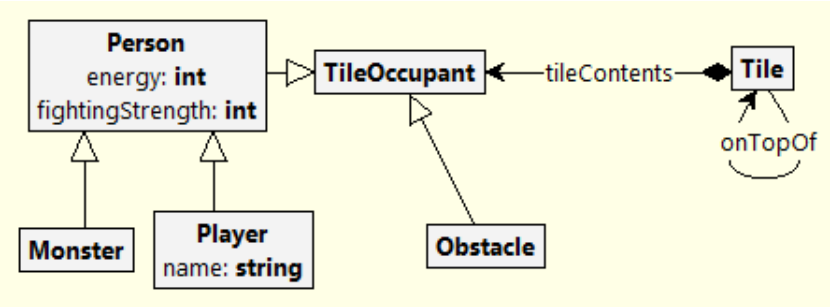

Figure 2: Groove metamodel

2.2. Instance models

Instance models are the start graphs(initial state of the system) the transformations work on and from which the LTS graph will be constructed.

2.2.1. AToMPM

In AToMPM instance models are instances of their respective meta-model. They are constructed using the defined concrete syntax and are DSL-specific. An example instance model is given in Fig.3.

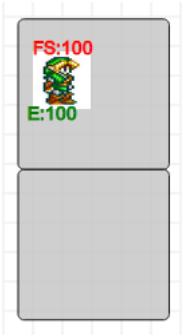

Figure 3: AToMPM model instance

2.2.2. Groove

Groove's state graphs are equivalent to AToMPM instance models, but they lack the concrete visual syntax that makes AToMPM modelling modeller-friendly. The example state graph is given in Fig.4.

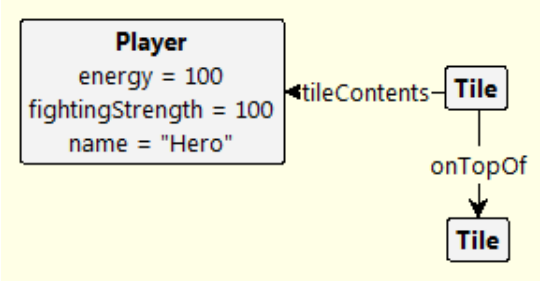

Figure 4: Groove model instance

2.3. Transformation rules

Transformations are the core of all GTSs and define, together with rule scheduling, the operational semantics of the DSL. Graph transformations consist of a left-hand side(LHS) that defines a pattern to match in the host graph, one or more negative application conditions(NAC) defining patterns that should not be matched and a right-hand side(RHS) defining what the LHS should be replaced with after the transformation. The example transformation rule used here moves the Player character from one tile to the tile below it, given that there is not already another occupant.

2.3.1. AToMPM

Graph transformation modelling in AToMPM is done with MoTiF(Syriani and Vangheluwe, 2013), using a pattern metamodel generated from the abstract and concrete syntax of a DSL. In MoTiF, the LHS, NACs and RHS are separate and linked through node and edge id's. Pivots(global names associated with a node) can be used to select certain specific nodes in the host graph, instead of matching any node with the same type. It is also possible to a condition on the matching of the LHS and NACs with inserted python code. Actions that change attributes of model elements can be written in the RHS using python as well. The example MoTiF transformation rule is given in Fig.5.

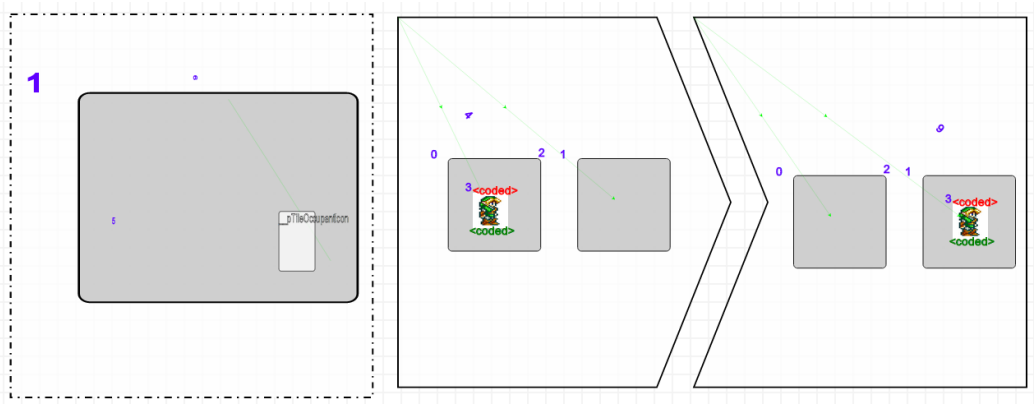

Figure 5: AToMPM transformation rule

2.3.2. Groove

In contrast to MoTiF, Groove uses a combined notation for LHS,NACs and RHS. The difference between elements of these three components is shown through visual queues. Groove has the notion of rule parameters, which are equivalent with MoTiF pivots. Groove transformations are more powerful than MoTiF transformations because of the ability to use existential qualifiers for nested rules and regular expressions for edge traversal. The latter can be emulated in MoTiF through python code manually traversing the instance graph, while the former is impossible to express. On the other hand, because of the arbitrary python code that can occur in MoTiF, there is another class of transformations that are impossible to express in Groove. Most occurrences of condition and action code can be expressed with Groove's product nodes, which can perform any operation on the basic types available. The example Groove transformation rule is given in Fig.6.

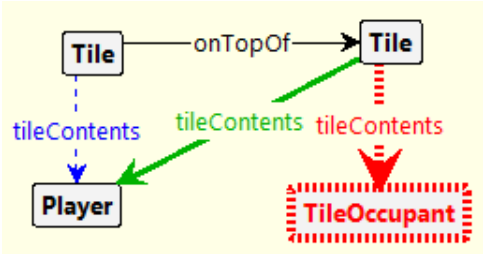

Figure 6: Groove transformation rule

2.4. Rule scheduling

Using transformations without a way of defining in what order they occur is not very powerful. Rule scheduling languages solve this. Although it is possible to emulate some form of scheduling inside the transformations themselves by adding extra information to the model, these languages offer a clear and concise solution.

2.4.1. AToMPM

MoTiF uses a graphical scheduling language consisting of rule blocks and success/failure edges between them. When a block succeeds in matching and executing its transformation(s), the success edge is followed to get the next rule block to be executed, otherwise the failure edge is followed. An example schedule is given in Fig.7. There are several types of scheduling blocks, each with their own semantics. These are explained in detail in (Syriani, 2011, p. 172). What follows is a quick summary:

1. Atomic rules

ARule Executes a rule once
QRule Checks if there is a match in the host graph.
FRule Executes a rule for all matches in the host graph
SRule Executes a rule as long as there are matches
CRule Starts execution of another schedule, allows for hierarchical rule scheduling

2. Composite rules

BRule Randomly executes one of its matching sub-rules.
LRule Loops over all matches of the base rule and executes the loop rule with the match of the base rule as implicit pivots.

2.4.2. Groove

Groove has two ways of explicit rule scheduling: rule priorities and control programs. A control program is specified in a simple imperative language, with rule applications as the basic syntactic blocks. Since it is an imperative language without recursion, this language is less powerful than the graphical control language of MoTiF. It does offer conditional looping, choice and simple function calls, which makes it able to emulate all but FRules and LRules.

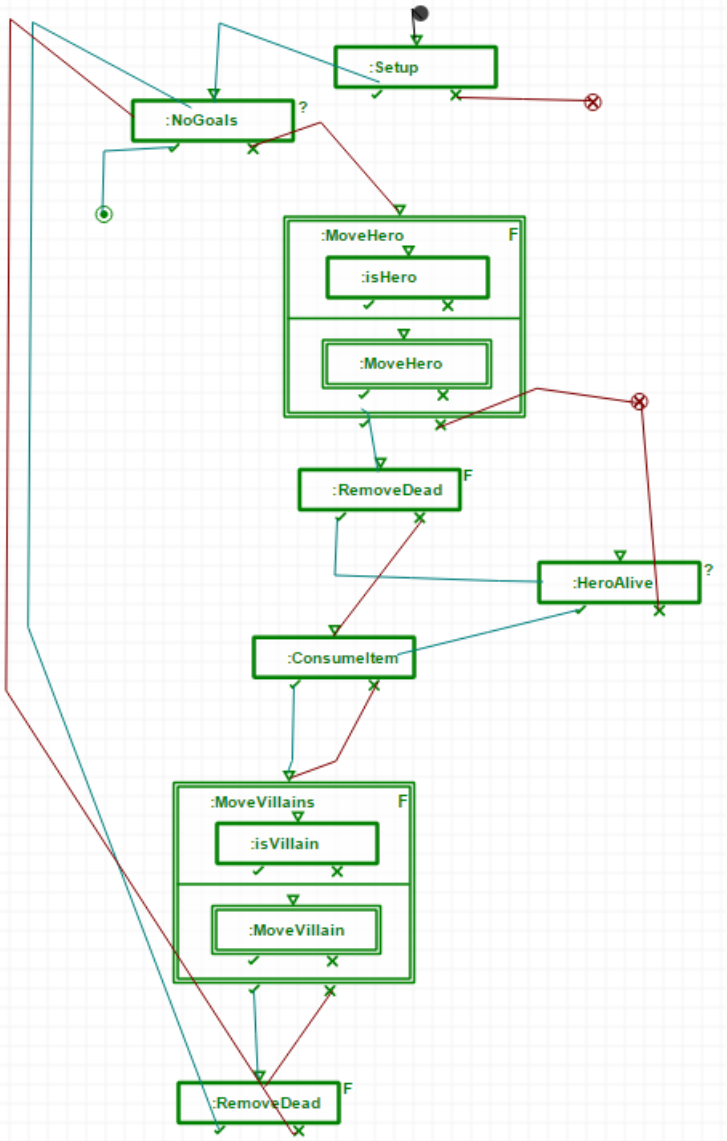

Figure 7: MoTiF schedule

3. Implementation overview

Our implementation heavily relies on AToMPM's metaDepth(De Lara and Guerra, 2010) exporter and metaDepth's code generation capabilities. We have created 4 EGL generators(main_m,main_mm,main_gt and main_r) and a script that simplifies running them, called run.sh. This script takes two parameters: the name of the mdepth file you are converting, and the name of the converter. Both names should be given without file extensions.

3.1. Meta-models

Meta-models can be converted to Groove type graphs by using the model export and running the main_mm generator. The meta-model, when exported as a model, should be exported with an "MM" suffix, to avoid a name clash with the export as meta-model. Since in AToMPM, edges can have attributes, whereas they cannot in Groove, we decided to transform AToMPM edges to Groove nodes, with In and Out edges to specify in which direction the edge goes.

A note on attributes: currently, only Groove-compatible attributes are supported(integers, booleans, strings and real numbers). Other attributes get ignored in the transformation process.

A final implementation detail is that we make all types inherit from an imported ModelElement type, to make pivots(3.3.2) possible.

3.2. Models

Models, just like meta-models, have a one-to-one mapping to Groove, which makes this conversion step consist of exporting the meta-model as a meta-model, exporting the model and running the main_m generator.

3.3. Transformations

Since there is no simple one-to-one mapping of transformations in AToMPM to rules in Groove, we implemented a Groove formalism in AToMPM, to which we transform the pattern contents of the transformation rule. This can then be exported to metaDepth and converted using the main_r generator.

3.3.1. AToMPM pattern meta-models

Pattern contents in AToMPM are instances of a pattern meta-model, which is generated from the abstract and concrete syntax of the DSL that is being transformed. This process can be done(and should be to be able to use this framework) through the RAMification transformations. Generating a pattern meta-meta-model is possible, when treating the pattern meta-model and its concrete syntax in the same way, which enables higher-order transformations. There is, however, no generic pattern meta-meta-model, to which all pattern meta-models conform. Thus, it is not possible to write higher-order transformations that transform any given pattern.

We solved this problem by creating this meta-model ourselves, in the form of the GenericTransform formalism. The elements of this formalism capture all relevant information of pattern instances. We then implemented a set of transformations that transform pattern nodes, edges and self-edges to their equivalent in the GenericTransform formalism.

Since these transformations are meta-model-specific, we created a set of EGL generators that can be started with the main_gt generator which runs on the meta-model(exported as model). These generate all necessary rules and two scheduling models. One that transforms to the GenericTransform formalism and one that chains the first with the Groove fromGT transformation. These generated transformations should be placed in a genericTransform subfolder of the formalism they belong to. Running the conversion on a transformation rule can be started with the start button on the Groove toolbar. The use of this button is required(see Section 3.4). When the conversion has finished you can export it to metaDepth using the export button on the same toolbar. This automatically unloads all non-Groove formalisms and runs the export to metaDepth.

3.3.2. Pivots

In AToMPM, pivots match only the element they're bonded with or are ignored when not already specified. To implement this, we made a Pivot type in Groove(Fig. 8), which then get checked or created in rules that specify input/output pivots. The example rule(Fig. 9) has an input pivot called 'hero' and an output pivot called 'eaten' on the Hero element.

3.3.3. Matching algorithm differences

The matching algorithms used in AToMPM and Groove work in subtly different ways. The two major differences are node merging and subtype

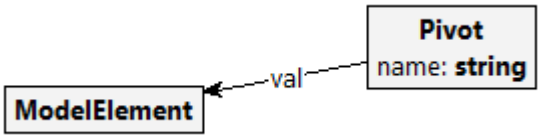

Figure 8: Pivot typegraph

matching. Node merging is a phenomenon in Groove where, if there are two nodes in a LHS of compatible types, they can represent the same node in the state graph after matching. Since this doesn't happen in AToMPM, it should also not happen in our generated rules. To disable this merging, we added inequality edges between all elements of the same pattern that could possibly merge. Subtype matching is optional in AToMPM transformations.

3.3.4. Queries

If a rule is meant to be used as a QRule, and thus has no RHS, we prevent all elements from becoming deleter nodes. If you need a query version of a rule, you need to save it as a separate rule lacking the RHS and use that rule in the QRule query attribute.

3.3.5. NAC

In AToMPM, a NAC works the same way as a LHS during matching, but if a NAC matches, the rule can't be applied. In Groove, each sub-graph of connected 'not:'-qualified nodes represents a single NAC. This presents a problem when there are non-connected nodes in an AToMPM NAC, since they wouldn't need to be matched together to cause a failed match in Groove when performing a naive transformation. Luckily, because of our disabling of node merging(3.3.3), all elements of a NAC are connected through at least the inequality edges, which makes them all match together as a single NAC. Links to LHS elements are handled through merger edges, which enforce two rule nodes to represent the same node in the state graph.

3.3.6. Condition and Action code

In AToMPM, it is possible to execute arbitrary python code to set constraints on values or calculate the new values of attributes. Groove rules can also manipulate attributes, but only in a more restricted way through product nodes(which support all basic operations on Groove's data types). Our solution includes support for most of these operations and data types. We

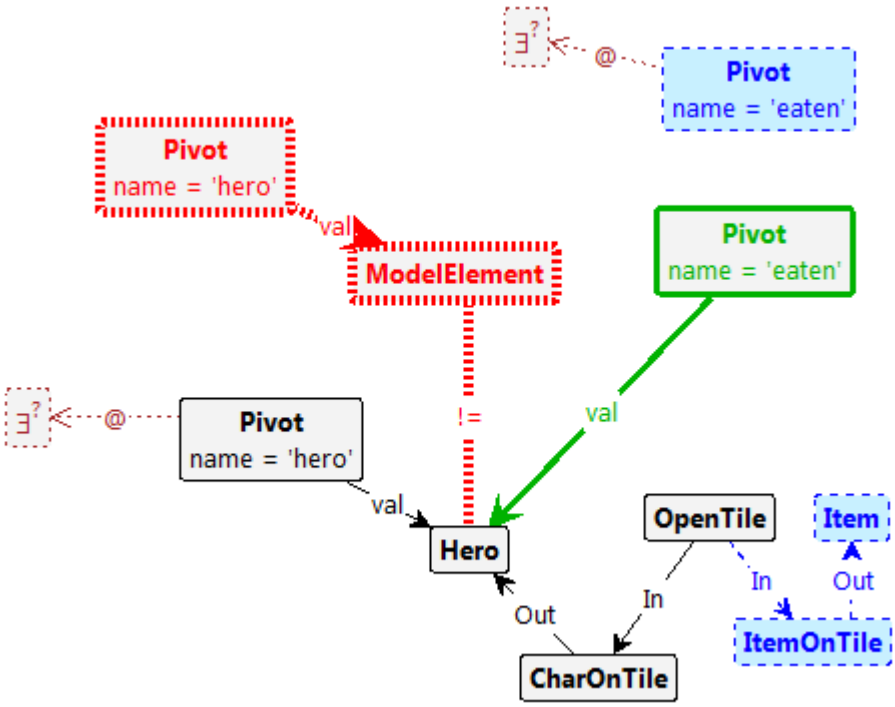

Figure 9: Example rule with pivots

emulate the environment the condition/action code runs in and insert our own version of the getAttr and setAttr methods. Our version of getAttr returns a custom object that will start building an expression tree when there are expressions applied to it. We then recurse down that tree and build the product node hierarchy. This approach sadly fails to work with boolean negation because of the way the '!' operator in python works. If there is any control flow logic that depends on values of certain attributes, the conversion will also fail, since these are cases that are almost always incompatible with the way Groove rules work.

3.4. Rule scheduling

Control programs in Groove aren't as powerful as the graph control language used by MoTiF. To make MoTiF scheduling possible in Groove, we opted to add the transformation schedule to the state graph, and implement the scheduling semantics in a control program(Fig. 10). Each transformation rule of the system checks whether a rule block with its own name is

```
1   MoTif.start;
2   while(true) {
3     try MoTif.BRule; //Expand BRules
4     try {
5       rules.any;
6       try MoTif.SRuleSuccess;
7     }
8     //Unroll until we can follow a success/fail edge.
9     until(MoTif.success|MoTif.recurseSuccess|MoTif.fail|MoTif.recurseFail) {
10      node current;
11      MoTif.unroll(out current)|MoTif.unrollRecurse(out current);
12      //Add recursive flag on BSRules
13      try MoTif.isBSRule(current);
14    }
15  }
```

Figure 10: MoTiF control

active in addition to matching its LHS/NAC. Since it's impossible to get the current filename in a MoTiF transformation, we add a string Groove node to the transformation when the Groove button is pressed before starting the conversion. FRule blocks are modelled through an auxiliary rule using an existential forall qualifier. This rule can be generated from an existing rule model with the Groove forall transformation, which adds a top-level existential qualifier to the rule elements. BRule blocks are simply a choice between multiple branches, we make sure all branches get the possibility to match and Groove will explore them all(see MoTif.BRule). LRule blocks are impossible to emulate without analysis of both the base and loop rules, because we would need to find which patterns in the base rule correspond to patterns in the loop rule to know what nodes we need to have the selection mechanism work on.

4. Future work

CRules are still unsupported because of time constraints, but we believe that it is possible to implement within the current framework.

The condition/action code parser could be made more robust, take into

account control flow(conditional testing may need the rule to be split into multiple versions).

A lot more can be done to make this conversion more user-friendly, as it now still consists of several switches between AToMPM and using the command-line to run EGL exporters(after which you need to manually move the files to the right locations).

5. Conclusion

We have given an overview of both AToMPM and Groove functionality and compared them for the purposes of modelling transformation systems. We conclude that while AToMPM excels at visual modelling, Groove is more adept at performing model checking on the system. We then described a way to convert AToMPM (meta-)models, transformations and rule scheduler to their Groove equivalents. Model checking arbitrary constraints can then be done by modelling a query that finds an acceptor/rejector state in AToMPM as a query transformation, exporting this to Groove and using it as the acceptor/rejector rule for Groove's model checking tool-set.

References

De Lara, J., Guerra, E., 2010. Deep meta-modelling with metadepth, in: Objects, Models, Components, Patterns. Springer, pp. 1–20.

Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V., 2012. Modelling and analysis using groove. International journal on software tools for technology transfer 14, 15–40.

Syriani, E., 2011. A multi-paradigm foundation for model transformation language engineering. Ph.D. thesis. McGill University.

Syriani, E., Vangheluwe, H., 2013. A modular timed graph transformation language for simulation-based design. Software & Systems Modeling 12, 387–414. URL: http://dx.doi.org/10.1007/s10270-011-0205-0, doi:10.1007/s10270-011-0205-0.