Electronic Communications of the EASST
Volume 54 (2012)

Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)

Adding Rule-Based Model Transformation to Modelling Languages in MetaEdit+

Simon Van Mierlo and Hans Vangheluwe

12 pages

Guest Editors: Christian Krause, Bernhard Westfechtel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Adding Rule-Based Model Transformation to Modelling Languages in MetaEdit+

Simon Van Mierlo[1] **and Hans Vangheluwe**[2]

[1] simon.vanmierlo@student.ua.ac.be
University of Antwerp, Belgium

[2] hv@cs.mcgill.ca
University of Antwerp, Belgium
McGill University, Montréal, Canada

Abstract: `MetaEdit+`is a commercial tool by MetaCase for creating domain-specific, syntax-directed visual modelling environments. `MetaEdit+`synthesizes such environments from user-provided metamodels and contains a Generator Editor for code/report generation. An API is provided to allow external manipulation of models through SOAP. Currently, the `MetaEdit+` tool does not natively support rule-based model-to-model transformation. Such transformations are useful as they allow domain experts to intuitively (using domain-specific notations) model either operational semantics (a simulator) or denotational semantics (through model-to-model transformation onto a model in a known formalism) of a modelling language. We will demonstrate how to add rule-based operational semantics to modelling languages in `MetaEdit+`. In our approach, transformation rules are visually created in `MetaEdit+`. The rule editor is synthesized using modified versions of the original language's metamodel. This modification is performed in a structured fashion using a process called RAMification. Both the model and the rules are exported from `MetaEdit+` to Python code. This code is combined with Py-T-Core, our library of transformation language primitives, to apply the rules on the model. Our demonstration has a client-server architecture, with the `MetaEdit+` visual modelling environment as the client and the transformation engine as the server. After each transformation step, in-place changes to the model are propagated to `MetaEdit+` for visualization using the SOAP API. A simple (manufacturing) Production System modelling language is used as an example.

Keywords: Model-Driven Engineering, Modelling Languages, MetaEdit+, Rule-Based Model Transformation

1 Introduction

Model-to-model transformations are an essential part of the Model Driven Engineering (MDE) approach. They allow the modeller to, for example, define semantics for a modelling language, either by simulating the model (operational semantics) or by mapping onto a known formalism (denotational semantics). These transformations can be defined in a number of ways. One of the most common and intuitive approaches is to use transformation rules. In ATL [JABK08],

the modeller textually defines transformation rules. This approach closely resembles programming, and may not be desirable for some purposes. The ability to use domain-specific notations in visual modelling languages is one of their main advantages, and this advantage is partly lost when using generic specificiations of transformations. Hence, using domain-specific notations (concrete syntax) of a visual modelling language to create models of transformations (and in particular, of transformation rules) lowers the threshold to the effective use of transformation. In addition to specifying individual transformation rules, the order in which the rules are tried may also be specified using a scheduling language. In Fujaba [NNZ00] and AToM^3 [LV02], amongst others, visual modelling of transformations is supported.

In this paper, we show how to add the ability to visually model and execute model transformations for modelling languages created in `MetaEdit+`[1]. The particular example we will focus on is the description of operational semantics of modelling languages. Operational semantics are a concrete application of inplace transformations which are used to give an operational meaning to models created in the modelling language. `MetaEdit+` currently does not natively provide support for model-to-model transformation. It does provide an API for externally manipulating models through SOAP, and a generator editor for code/text generation. We will use these two facilities to (1) enable the modelling of transformation rules inside `MetaEdit+`, to (2) export a rule-based model transformation model constructed in `MetaEdit+` to Python and to (3) subsequently execute it, using our library of graph transformation primitives T-Core [SV10]. After executing the graph transformation, the results of rewriting the model (which is in essence a typed, attributed graph), are propagated back to `MetaEdit+` for visual feedback. A similar method was used in constructing the tool `booggie` [HSH09] for rule-based design-space exploration. There, a visual environment was constructed around the efficient graph rewriting kernel GrGen.NET [JBK10]. Our demonstrates the practical applicability of RAMification [KMS+10] and is structured in a way which makes it easy to reproduce the results in `MetaEdit+` or any other tool that provides similar functionality.

The paper is structured as follows. Section 2 explains the running example of the paper, which is a domain-specific modelling language for Production Systems we would like to build a simulator for. Section 3 explains the process of RAMification and how it will be used throughout the paper. Section 4 explains how the rule editor in `MetaEdit+` is constructed using RAMification, starting from the original metamodel of a (domain-specific) language. Section 5 presents the architecture of our solution. Section 6 takes a closer look at the execution flow when a rule is executed and Section 7 concludes and suggests future work.

2 Running Example

To demonstrate the approach used in this paper we will define a domain-specific language which is easy to understand but nevertheless non-trivial. The language we create is used to model production systems of Armoured Personnel Carriers (APCs). The APCs are constructed using different parts which are generated by machines at the start of the production line. There are

[1] http://www.metacase.com/

five different kinds of parts: wheels, bodies, tracks, water cannons and machine guns. Different amounts of these parts are needed to assemble two types of APC: war APCs and riot APCs. There are four types of machines: generators, assemblers, quality control machines and a sink. A machine has to be operated by an operator in order to perform its task. An operator can only operate one machine at any point in time. Machines are connected to each other by conveyor belts. A generator has one outgoing conveyor belt on which generated parts are placed. Assemblers have an incoming conveyor belt which supplies parts and an outgoing conveyor belt on which finished products are placed. Quality control machines have an incoming conveyor belt which supplies finished products that can either be broken or functioning correctly and two outgoing conveyor belts which are used to distinguish between these two types. Repair machines have one incoming conveyor belt, which supplies broken finished products and one outgoing conveyor belt, where repaired products are placed. Sinks have one incoming conveyor belt which supplies finished products. Conveyor belts can have multiple incoming connections (either from conveyor belts or from machines) and one outgoing connection (either to a conveyor belt or to a machine). This explains, informally, the syntax and static semantics (welformedness rules) of the example language. These rules can be formalized in a user-defined metamodel in `MetaEdit+` and models can be visually created in the synthetized environment.

Once models can be created, we typically want to give them meaning. In this case, the meaning will allow for simulation of the dynamics of a model. A set of semantic rules will be created for our modelling language, defining how parts are generated, how they move from one conveyor belt to the other, which parts are taken off from which conveyor belt and at what time to create APCs. Constraints could be added to the language, enforcing that an operator is present at a machine in order for it to work. Using these semantic rules, simulation experiments can be created. Running these experiments allow a modeller to discover interesting facts about particular production system models. Subsequently, performance metricus such as troughput of model variants can be evaluated to design an optimal production system.

The creation of this simulator for our production system modelling language is done by defining a sequence of endogenous transformations that describe how the state of a running production system gets updated as time progresses. This simulator corresponds to the operational semantics of our language. This is different from denotational semantics, where a model in one language is mapped by an exogenous model-to-model transformation onto a formalism with known semantics (such as Petri Nets or code). The techiques we describe in this paper can also be used to define denotational semantics.

Defining our production system simulator can be done in a number of ways. An external simulator could be written in a general-purpose programming language and production system models are exported by a `MetaEdit+` exporter to be simulated by that program. This is, however, not in line with the MDE approach, where as much as possible should be modelled explicitely, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). In the next section, we take a look at the possibilities to explicitely model transformations using a systematic approach.

3 RAMification

In [KMS+10], Kühne et al. advocate the explicit modelling of abstract and concrete syntax of transformation languages. They state that the advantages of metamodelling in general apply to the modelling of transformation language in particular as well: (1) the specification is not hidden in the code of a tool, making it easier to understand and correct, (2) one can reason about the specifications and the instance models they describes, (3) one may synthesize modelling environments from the specification and (4) this makes it easy for users to alter the specification instead of requiring a new tool release. That is why Kühne et al. explore the possibility of explicitely modelling (visual) rule-based model transformation languages for (visual) modelling languages described by a metamodel. The rules consist of a Left-Hand Side (transformation pre-condition) pattern (LHS, describing the part of the model that should be matched for the rule to be applied), zero or more Negative Application (transformation pre-)Conditions (NACs, specifying the patterns that, when found, should stop the rule from being applied), and a Right-Hand Side (post-condition) pattern (RHS, specifying how the matched part of the model should be rewritten).

The patterns that can appear in the LHS, RHS and NACs are, of course, very similar to the models we can create in the original modelling language. It is therefore logical to try to reuse the metamodel that defines the original language for the pattern specification language, instead of creating one from scratch. Above all, starting from the (domain-specific) modelling language's metamodel allows for a highly specific transformation language which only permits transformation rules with patterns specific to the modelling language, including language-specific (visual) pattern notations.
We cannot simply copy the metamodels and use them for specifying the patterns of a rule. Firstly, the patterns that appear in rules are not necessarily well-formed models in the original modelling language. For instance, for the Production System language it may be useful to be able to specify a pattern which contains a conveyor belt that has no outgoing connections. In the original language this is not a well-formed model as each conveyor belt should have an outgoing connection to either another conveyor belt or to a machine. In order for the pattern specification language to be useful, these well-formedness rules should be *relaxed*. Secondly, a number of elements have to be added to the metamodel. It should be possible, for example, to identify model elements across the LHS, RHS and NACs. This is typically done by *augmenting* the metamodel, adding labels to entities in the LHS, RHS and NAC patterns. Thirdly, the data type of model element properties should be *modified* as to allow the definition of constraints on properties as well as actions to compute the new value of a property. These are the three main concepts of RAMification: Relaxation, Augmentation and Modification. The authors of the paper describe a semi-automatic process which a developer can follow to create a customized pattern language with minimal effort, starting from the original metamodel of the language.

In the next section, we explore how RAMification is used to create a transformation language specific to our example language. We do this specifically in the commercial tool `MetaEdit+` and thus show the applicability of RAMification in practice, beyond its demonstration in the research tool AToM^3.

4 The Rule Editor

A rule editor is an interactive (in our case visual) environment for creating model transformation rules. Thisis the first essential part of our implementation we will explain. The model transformation system we want to construct consist of a number of rules, which can be applied to a model. To make the rule-modelling environment as domain-expert-friendly as possible, these rules should re-use the domain-specific visual notation of the elements to be transformed. Rules consist of three parts: exaclty one left hand side (LHS), exactly one right hand side (RHS) and zero or more negative application conditions (NACs). The LHS holds a pattern to indicate which part of the model is to be matched. If the rule is tried and a match is found for the LHS, the transformation engine will also try to find a match for the NACs. If a match for one of the NACs is found, the rule will not be executed. If no NAC match can be found, the rule will rewrite the model by replacing the elements found by matching the LHS by the the corresponding elements described in the RHS. In general, a transformation rule can transform elements present in the LHS of the rule to elements of aribitrary modelling languages. These are called exogenous transformations. In this paper, we restrict ourselves to the modelling of operational semantics which merely updates the state of a model. Hence, only elements of the modelling language we are creating semantics for will appear in rules. The transformations used are thus endogenous and in-place. Note that our technique also works for exogenous transformations.

The rule editor makes use of the decomposition capabilities of `MetaEdit+`. An object of one metamodel can, in `MetaEdit+`, be decomposed into a graph conforming to another metamodel. This enables us to create a layered structure for graph transformation models. Figure 1 depicts a model of a graph transformation language. We will focus on the particular case of (layered) graph transformations (or, if they are used to define languages, grammars). A transformation has a name and consists of a number of rules. The rule objects decompose into graphs conforming to the rule metamodel, which consists of the three elements mentioned above: one LHS, one RHS and zero or more NACs. A rule also has a name and a precedence, which is a positive integer. The precedence defines layers in the transformation. The transformation will, while it is executing, choose a rule at random from the currently executing layer. Once none of the rules in the current layer can be executed, the execution of rules proceeds to the next layer. As we wish to use model transformation for simulation (and hence is in principle non-terminating), our semantics loops back to the first layer once no more rules can be fired in the last layer. The LHS and NAC objects of a rule decompose into graphs conforming to the pre-condition pattern metamodel. The right hand side object of a rule decomposes into a graph conforming to the post-condition pattern metamodel. We create these metamodels starting from the original metamodel and apply RAMification on them. In particular, these are the steps we took to create the modified versions of the metamodel:

1. First, make two copies of the original metamodel: one for the pre-condition pattern language (NAC and LHS) and one for the post-condition pattern language (RHS). When we refer to "the metamodel" in the next steps, we mean one of these copies and not the original metamodel.

2. Relax the constraints on the metamodel's well-formedness. A rule often only matches a

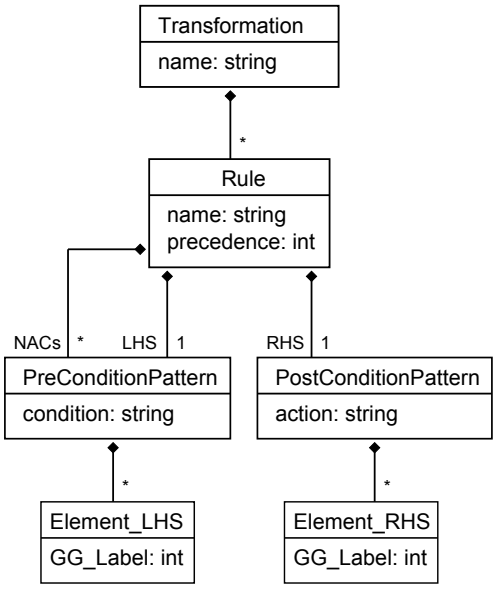

Figure 1: Structure of a transformation. Adapted from [KMS+10].

part of a model and this may not be a well-formed model conforming to the original metamodel. It is also possible for abstract superclasses to appear in rules, which is impossible in the original modelling language. For these abstract classes, a default visual concrete syntax is created which enables the modeller to create them in patterns.

3. Append the suffix `_LHS` (pre-condition pattern) or `_RHS` (post-condition pattern) to the class names of the objects and relationships.

4. Add a property called `GG_Label` of type "Number" to each object and relationship. This property is used by the graph matcher to identify nodes across the different parts of a rule.

5. Append the suffix `_LHS` (pre-condition pattern) or `_RHS` (post-condition pattern) to each property of an object or a relationship and change its datatype to "String". The properties now define a condition (pre-condition) or an action (postconsition pattern) instead of an actual value. These strings are, in this case, Python executable code that has to evaluate to a Boolean value in case of a condition, or to the new value of the property in case of an action.

6. Add a property called "constraint" (pre-condition pattern) or "action" (post-condition pattern) to the metamodel. These represent, respectively, the condition that has to be satisfied before a rule can be executed and the action that has to be taken after the rule has executed.

In order to make simulation possible, the original metamodel has to be modified as well. As the layered architecture of the transformation gives priority to layers with a lower precedence value, mechanisms to ensure *fairness* have to be implemented. This is achieved by adding properties to objects that are modified by rules. These properties can be checked in the NAC(s) or LHS of a

rule: only when a particular value is found can the rule be executed. To disable the rule, the RHS sets the property to anything else than that value. In our example transformation, for instance, we have added a "moved" property to the "Operator" object which has to have the value 0 in order for the rule "MoveOperator" (which moves an operator from one machine to another one) to be applied. The RHS sets the value of this property to 1 which effectively disables the rule, preventing the rule from executing continuously. The top layer of our example transformation consists of rules that set these properties back to their initial values so all rules become enabled again. Thus, each pass through all the rules only considers each match for each rule once and fairness is achieved.

The rule editor has to be constrained in a way such that only valid rules can be created. While both the pre- and postcondition pattern languages allow the instantiation of (originally) abstract classes, a constraint has to be defined on the RHS of rules, disallowing instances of abstract classes to be instantiated without a corresponding instance (i.e. with the same GG_Label) in the LHS. Otherwise, the execution of a rule could attempt to instantiate an instance of an abstract class, which is prohibited in the original language. In the case where a corresponding instance exists in the LHS, the execution of the rule would match and rewrite a concrete subclass in the model which is being transformed, which is a valid operation.

Note that while in our prototype tool AToMPM [Man12], we perform RAMification fully automatically, the RAMification described in this paper was done manually inside `MetaEdit+`.

5 Architecture

In this section, additional elements present in `MetaEdit+` or implemented in Python that are important to our solution will be discussed. In Figure 2 an overview of the architecture of our solution is shown. This is a client-server architecture: the transformation engine acts as the server, `MetaEdit+` as the client.

5.1 Python: Abstract Syntax Graph (ASG)

An abstract representation of `MetaEdit+` models was created in Python. This component has two functions: it provides a data structure to export models to using the `MetaEdit+` exporters, and it acts as an abstraction layer for the SOAP API. All methods defined on this structure make use of the SOAP API to reflect changes visually in the `MetaEdit+` model. These classes are as generic as possible. It is therefore possible to export any type of `MetaEdit+` model to this Python structure.

5.2 `MetaEdit+`: API and Generators

The SOAP API of `MetaEdit+` is heavily used in our solution. It provides methods to query and update models, which are used by the ASG component in Python.
The generator editor facility of `MetaEdit+` was used to create two types of generator: one for models, and one for rules. As we saw in Section 4, a rule consists of exactly one LHS, zero or

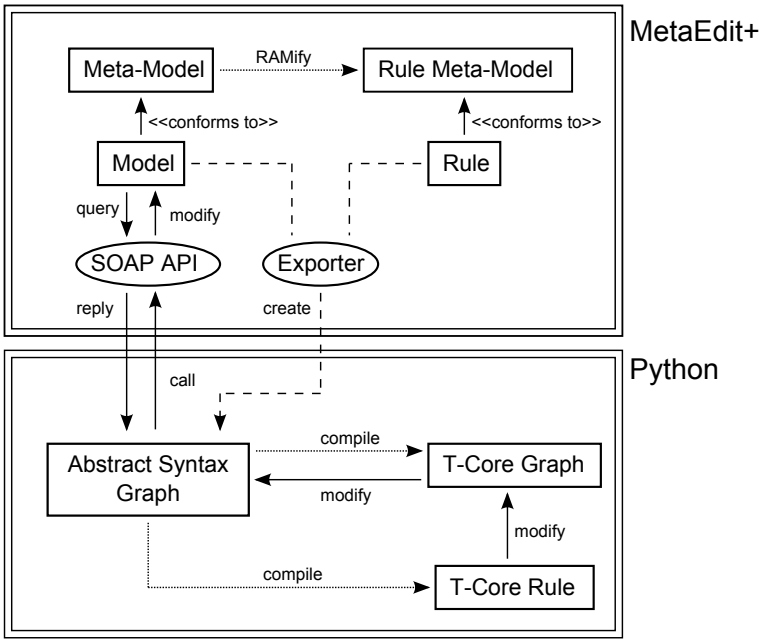

Figure 2: Architecture, including calls and relations between different components.

more NACs and exactly one RHS. These components of a rule are, like models, mapped to the ASG structure in Python.

5.3 T-Core: Graph Rewriting

T-Core is a library of graph transformation primitives [SV10]. It is used in conjunction with a scheduling language, which in our case is Python. We only need a small subset of T-Core: the ARule (Atomic Rule), which chooses one match of the set of all matches (matching the LHS, considering the NACs) and transforms the LHS to the RHS. Before T-Core can be used, the following challenges have to be dealt with.

T-Core has its own data structures for graphs and rules. A compiler was built to compile the ASG representation into a T-Core graph (for models) or a T-Core rule (for rules).

To compile an ASG of a `MetaEdit+` model, the compiler iterates over all nodes in the ASG twice. During the first pass, it adds all nodes to the graph. This includes both object nodes and relationship nodes. When adding a node, it copies all properties of the source node to the target T-Core node and adds an attribute to the T-Core node which will be used to identify it in the source ASG. This attribute will be used when T-Core has executed a rule on its graph representation, as the changes have to be propagated back to the ASG (see Section 6). It should be ignored by T-Core in the matching phase as it is not a property of the corresponding object in the model. T-Core provides a mechanism to achieve this by naming the property in a particular way. During the second pass, edges are added from relationship nodes to their source and target nodes. These are not present in the source graph, but are needed by T-Core.

Compiling a rule is almost identical. First, a T-Core rule object is instantiated. Then, the LHS, RHS and NACs are compiled as outlined above (as they are represented by an ASG as well, since they are a special kind of model written in the modelling language) and added to the rule. However, an extra step has to be taken for the attributes of these nodes. In the LHS, RHS and NACs, T-Core expects the properties to be functions. Properties in the LHS and NACs have to return a Boolean value, properties in the RHS have to return a value which corresponds to the new value of that attribute. The strings that are given by the modeller for these properties are wrapped in functions that evaluate the string as Python code and return the result of this execution. The pattern condition for the LHS as well as the pattern action for the RHS are wrapped similarly. As both the model and rules are now represented as T-Core graphs, the rules can be executed.

6 Executing Rules

We now have all elements needed to create a set of rules in `MetaEdit+` and execute them on a model. This section explains how a rule is applied on a model. We start by creating a rule in `MetaEdit+`, then export it to Python. There, it will be compiled to T-Core together with an exported model. T-Core will rewrite its graph and report back the changes, which will be used to modify the ASG accordingly.

6.1 Creating The Rule

We will consider the moving of an operator from one machine to another as an example rule. This rule is one of the rules that define the semantics of our example production system language. It contains all of the functionality we want to demonstrate but is still basic enough to be able to explain the general principles involved.

Figure 3 depicts the rule as it appears in the generated `MetaEdit+` editor. An operator in our language can be connected to either an assembler, a quality control machine or a repair machine. All three of these machines inherit from the processor abstract superclass. The LHS of the rule defines what should be matched: two processors, one of which the operator is connected to. A condition for the "moved" property, states that it should be equal to 0. As we do not want two operators connected to the same machine, we also define a NAC. In the NAC, the processor we want to move to (with `GG_Label` $= 4$) has an operator connected to it. By defining this NAC, we make sure that whenever the rule is executed no operator is connected to this processor. The RHS defines what the matched subgraph of the LHS should look like after executing the rule. Here, the relationship between the operator and the original processor has been removed, while a new one is created between the operator and the new processor. The "moved" property of the operator is set to 1, which ensures this rule is only executed once until it is reset back to 0.

6.2 Compiling and Executing The Rule

As explained in Section 5.3, the model and the three parts of the rule are compiled to T-Core structures. It is important to point out that subtype matching is used. Without subtype matching, the moving of an operator would have to be split into several rules, to include every possible

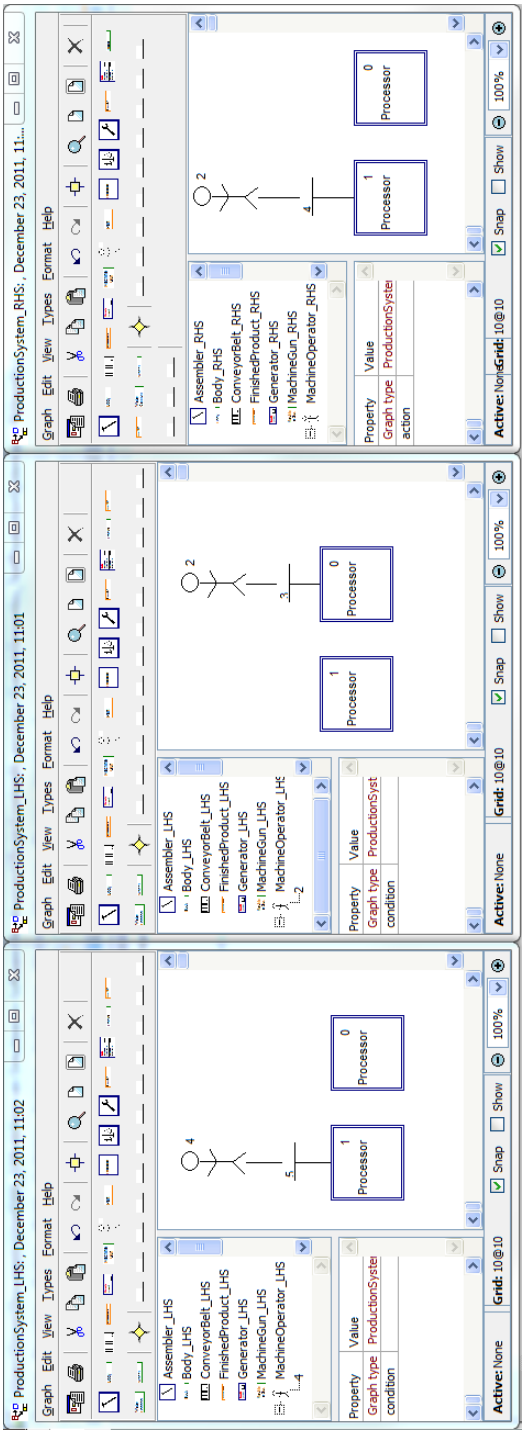

Figure 3: Example rule: operator moving from one processor to another. From left to right: NAC, LHS and RHS.

combination of processor classes. This would lead to an explosion in the number of rules. T-Core supports subtype matching: a list of subtypes for each type can be passed to T-Core. To execute the rule, the compiled rule (which is an ARule object) is given the T-Core representation of the model. T-Core will try to match the LHS and then choose one of the matches at random in case there is more than one. This randomness is controlled through the seeding of the internal random number generator of T-Core which ensures identical results (for experiment repeatability) when the transformations are run multiple times. Then, it will perform the necessary operations as defined by the RHS of the rule on this match. Internally, it changes its own representation of the model, and reports back a list of changes (an "edit script"). These changes include, but are not limited to, the changing of attributes, the creation or removal of nodes and the creation or removal of edges.

6.3 Modifiying The ASG

The list of changes made to the T-Core graph is subsequently used to modify the original ASG of the model. In the compilation process of the ASG, we made sure the nodes in the T-Core graph can be linked back to their original ASG nodes. This makes it possible to perform exactly the same changes to the ASG as were made to the T-Core graph which ensures both graphs represent the same model. In addition, the operations that change the ASG propagate these changes through the SOAP API to the original model in `MetaEdit+`, which results in visual feedback.

7 Conclusion and Future Work

In this paper, we have shown how to add operational semantics to languages created in `MetaEdit+`. First, a rule editor was created in `MetaEdit+` which allows us to visually create transformation rules which are combined in a (layered graph) transformation model. The transformation model was then exported to Python, where it can be executed on an exported `MetaEdit+` model using T-Core as a backend. The execution of a graph transformation results in a series of graph rewritings which visually propagate to the original `MetaEdit+` model by using the SOAP API of `MetaEdit+`.

Future work is outlined below.

- **Denotational Semantics of `MetaEdit+` Languages**: In this paper, we have added operational semantics to a (production system) modelling language. Further research will investigate adding denotational semantics to languages. The difference with the work described in this paper is that multiple metamodels have to be combined. In the rule editor, it should be possible to use concepts of the source language as well as the target language.

- **Automatic RAMification of Meta-models in `MetaEdit+`**: In `MetaEdit+`, metamodels of languages can be exported to and imported from XML. It should therefore be possible to automate the RAMification process of metamodels.

- **Other Environments**: The technique outlined in this paper could be used with other front- and backends. An example of this would be to add model-to-model transformations to the Eclipse Graphical Modelling Project[2], using for example the very efficient graph rewriting kernel GrGen.NET as backend.

Bibliography

[HSH09] B. Helms, K. Shea, F. Hoisl. A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. *ASME Conference Proceedings* 2009(49057):841–851, 2009.
doi:10.1115/DETC2009-86851
http://link.aip.org/link/abstract/ASMECP/v2009/i49057/p841/s1

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming, Special Issue on Second issue of experimental software and toolkits (EST)* 72(1-2):31–39, June 2008.

[JBK10] E. Jakumeit, S. Buchwald, M. Kroll. GrGen.NET - The expressive, convenient and fast graph rewrite system. *STTT* 12(3-4):263–271, 2010.

[KMS+10] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer. Explicit Transformation Modeling. In Ghosh (ed.), *Models in Software Engineering*. Lecture Notes in Computer Science 6002, pp. 240–255. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12261-3_23.
http://dx.doi.org/10.1007/978-3-642-12261-3_23

[LV02] J. de Lara, H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In Kutsche and Weber (eds.), *FASE'02*. LNCS 2306, pp. 174–188. Springer, Grenoble, France, April 2002.

[Man12] R. Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill University, June 2012.

[NNZ00] U. Nickel, J. Niere, A. Zündorf. The FUJABA environment. In *ICSE'00*. Pp. 742–745. ACM, Limerick (Ireland), June 2000.

[SV10] E. Syriani, H. Vangheluwe. De-/Re-constructing Model Transformation Languages. *Electronic Communications of the European Association of Software Science and Technology* 29, March 2010.

[Syr11] E. Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University, 2011.

[2] http://www.eclipse.org/modeling/gmp/