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

Simon Van Mierlo

University of Antwerp, Belgium

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/text 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+` . Rules are visually created in `MetaEdit+` . The rule editor is synthesized using modified versions of the original language's metamodel. This modification is done 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 execute 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 (manufacturing) Production System modelling language is used as an example.

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

1. Introduction

Domain-specific languages are used to create abstractions of a problem using concepts and notations that are bound to a specific domain. It allows not only developers, but also domain-experts to rapidly and intuitively create models to develop software. Domain-specific visual languages are a special kind of domain-specific language which provide a visual modelling interface. There are lots of tools supporting the creation of these visual modelling languages. Central to most of those is the concept of a metamodel, which defines the concepts of the language and the relationships that can exist between them. This metamodel defines the abstract syntax of the modelling language and models created in the language are said to conform to the metamodel of that language. In the case of visual modelling languages, the developer also has to define the graphical appearance of the concepts, which is called the concrete syntax of the modelling language. Using these two concepts, a model which belongs to the language can be visually created.
Model-to-model transformation is another essential concept 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 mapping onto a known formalism (denotational semantics). These transformations can be defined in a number of ways. In ATL [Jouault et al. (2008)], the modeller has to textually define the transformations. This approach closely resembles programming, and may not be desirable for some purposes. The domain-specific notation provided by visual modelling languages is one of their biggest advantages, and this advantage is partly lost using when using ATL and similar tools. Another option is to use the domain-specific notation of a visual modelling language to create rules that define transformation steps. By reusing the domain-specific notation, the threshold for effective use of transformations is lowered. To prescribe the order in which rules should be tried, they may be combined using some scheduling language. In Fujaba [Nickel et al. (2000)] and AToM^3 [de Lara and Vangheluwe (2002)], amongst others, this approach is used.
In this paper, we show how to add operational semantics to `MetaEdit+`[1]. `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

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

use these two facilities to export a rule-based model transformation model constructed in MetaEdit+ to Python and there execute it, using our graph transformation kernel T-Core [Syriani and Vangheluwe (2010)]. After executing the graph grammar, the results of rewriting the graph are propagated to MetaEdit+ for visual feedback. A similar method was used in constructing booggie [Helms et al. (2009)]. There, the developers have constructed a visual environment around the graph rewriting kernel GrGen.NET [Jakumeit et al. (2010)].

As a running example, we will use a production system formalism. This language defines production lines as a collection of machines, conveyor belts and operators. The production line manufactures armoured personnel carriers (APC's). The parts that are used to construct these APC's are generated at the start of the production line, after which they move on conveyor belts to an assembler. There, an APC is constructed by an operator. Subsequently, an APC is inspected in a quality control station and if needed repaired in a repair station, both operated by an operator. Finally, a sink removes the APC from the model and keeps count of the total amount of finished APC's.

The structure of this paper is as follows. Section 2 explains the process of creating a visual modelling language in MetaEdit+ . In Section 3, the construction of the rule editor in MetaEdit+ is explained, starting from the original metamodel of the language. Section 4 explains what the architecture of our solution looks like. Section 5 takes a closer look at the flow of execution when a rule is executed. Section 6 concludes and make suggestions for future work.

2. Creating a Language in MetaEdit+

In this section, the visual modelling environment for our example language is created. The first task is to define the static semantics of this language in MetaEdit. These are given by the following rules.

1. A production system is a set of connected machines and conveyor belts.
2. Machines can be operated by an operator.
3. A conveyor belt carries (unfinished) products.
4. Machines are connected to exactly one conveyor belt on which they can drop processed products.
5. Machines can either generate the parts of an APC (wheels, tracks, bodies, machine guns or water cannons), or process these parts or assembled APC's.

6. Machines that process can be assemble machines, quality control or repair machines.
7. Machines that process are connected to exactly one conveyor belt from which they take products that are to be processed.
8. Conveyor belts can be connected.
9. Quality control machines are a special kind of machine, as they can put products that need to be reworked on a different conveyor belt.

`MetaEdit+` metamodels are created in the meta-metamodelling language GOPRR. It is an acronym for the concepts it provides: Graph, Object, Property, Relationship and Role. A graph is the top-level element and represents the metamodel. A graph consists of objects and relationships. An object represents a concept in the visual modelling language (for example, an operator in the production system language). A relationship relates two or more objects with each other. Each object that is part of a relationship has a role associated with that relationship. Properties can be assigned to objects, relationships and graphs.
In Figure 1, the metamodel for the production system language is visually represented. The rectangles represent objects, diamonds relationships and circles roles. Multiple inhertiance is not supported, which is why assemblers, repair machines and generators all have a seperate relationship to an outgoing conveyor belt. If multiple inheritance would be supported, this relationship could be moved to a common superclass. Some of the properties will be further explained in Section 3.
It is possible to define constraints in the GOPRR language. There are two types of constraints: role connectivity constraints and relationship connectivity constraints. They define in how much roles or relationships a certain object can be in, respectively. It is only possible to define an upper bound, not a lower bound. Defining 'an Operator can be in at most one relationship of type OperatorToProcessor' is possible, but defining 'an Operator has to be in at least one relationship of type OperatorToProcessor' is impossible. Defining visual constraints, for instance saying that a part always has to be on top of the conveyor belt it is connected to, is not supported either. Figure 2 shows the interface for creating a relationship connectivity constraint.
Once the abstract syntax of the language is defined, a concrete visual representation has to be given to each entity. By default, each object is a rectangle with its name in it. To define a custom representation, the symbol editor can be used. Figure 3 shows how the symbol editor was used to create the

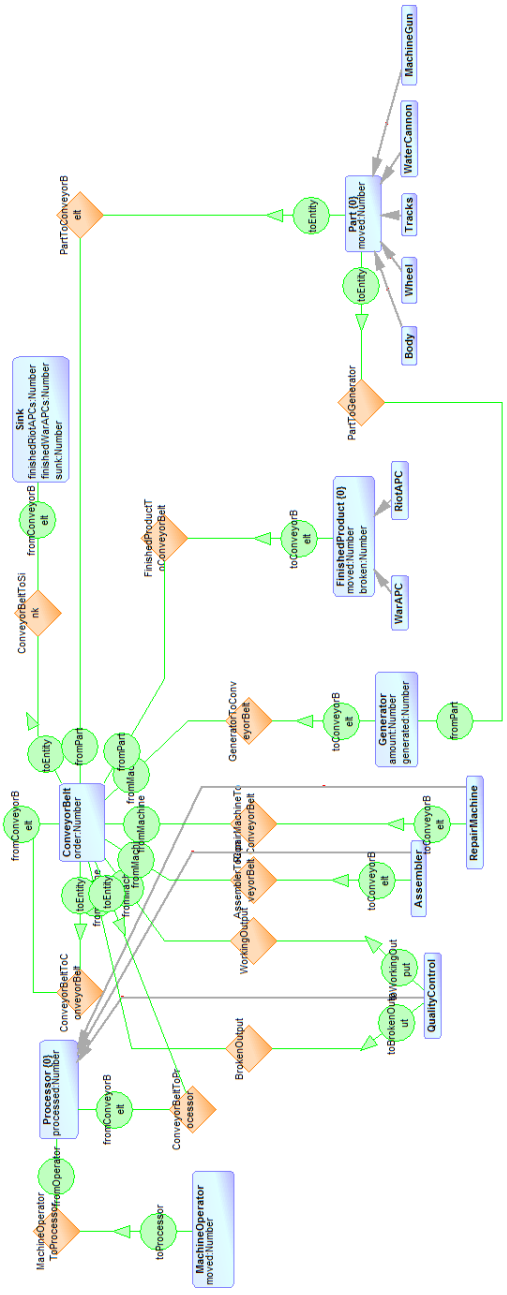

Figure 1: The metamodel for the production system language, as represented in the GOPRR meta-metamodelling language in `MetaEdit+` .

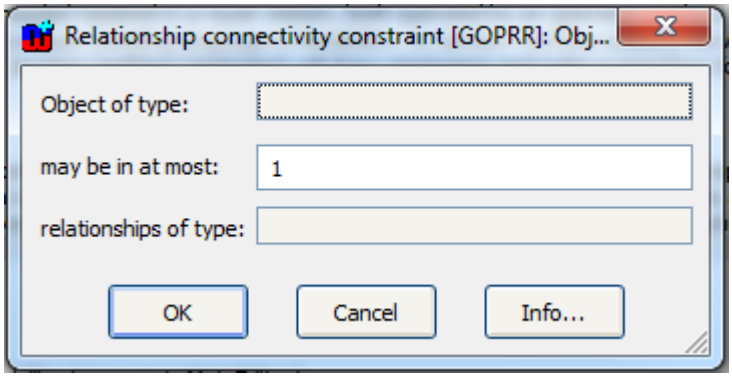

Figure 2: `MetaEdit+` provides support for defining certain types of constraints.

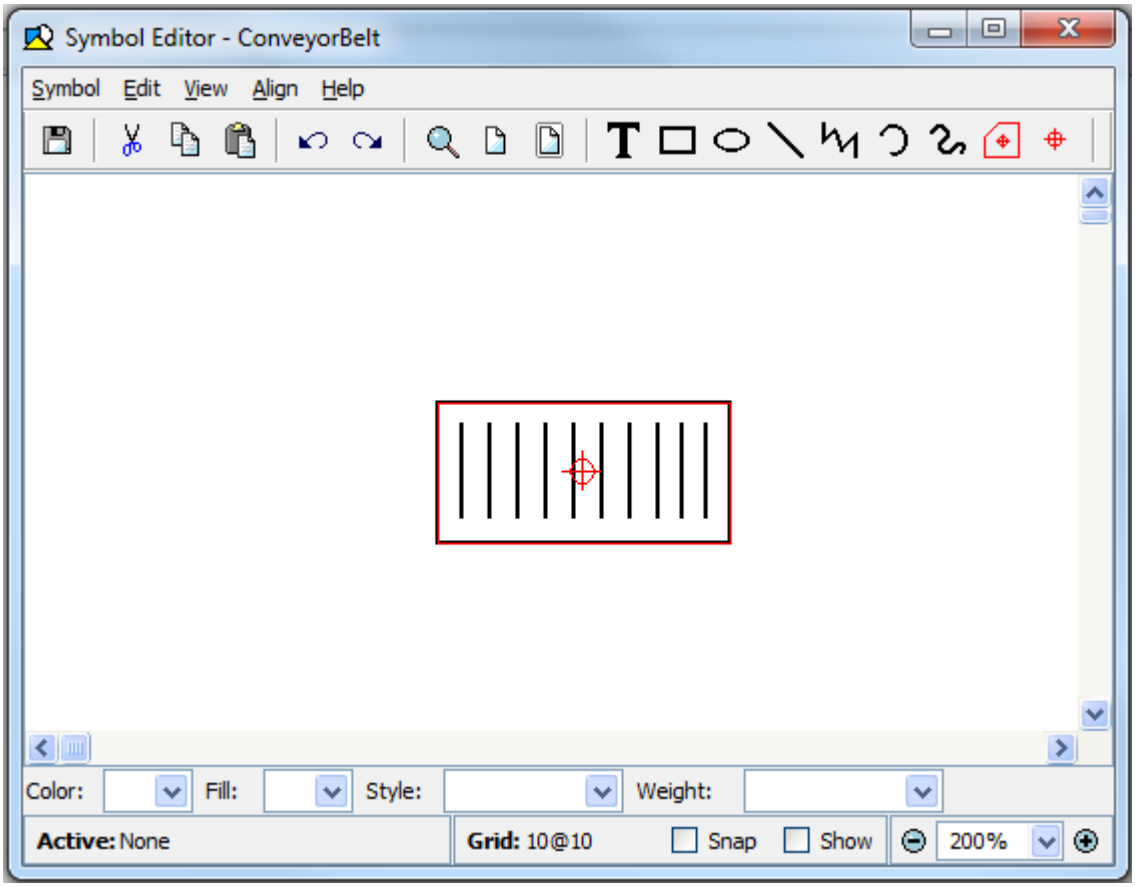

Figure 3: Defining the graphical appearance of a ConveyorBelt object.

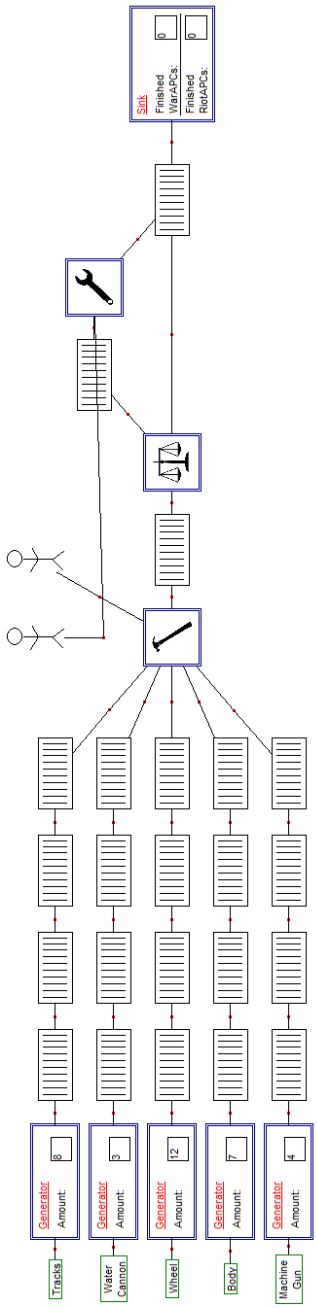

Figure 4: A sample model created with the production system visual modelling language.

visual representation of a conveyor belt. Optionally, icons for entities can be created. These icons will be used in the visual modelling environment for the language to make the buttons to create entities more visually appealing. The icon editor is very similar to the symbol editor. It is now possible to use the language we defined to create visual models of production systems. An example model is given in Figure 4.
Once the abstract and concrete syntax of the language is developped, it is time to add meaning, or semantics to it. The sample model in Figure 4 is meaningless until proper semantics are defined for it. In this paper, we will define operational semantics that can simulate a model using graph grammars consiting of rules. The semantics are given by the following rules.

1. There is only one direction in which products can be moved from one conveyor belt to the other.
2. Products can move from a conveyor belt to a connected conveyor belt.
3. A generator for a certain APC part can put such a part on its connected conveyor belt.
4. Machines can only process when there is an operator at the machine.
5. An operator can go from one machine to the other.
6. An assemble machine with an operator can take two tracks, a body and a machine gun from the input conveyor belt and process them into a war APC that is put on the output conveyor belt, or can take four wheels, a body and a water cannon from the input conveyor belt and process them into a riot APC that is put on the output conveyor belt.
7. A repair machine with an operator can take an (unfinished) product from the input conveyor belt, repair it, and put it on the output conveyor belt.
8. A quality control machine with an operator can take an assembled piece from the input conveyor belt and in case of a successful outcome puts it on the output conveyor belt, and in case of an error puts it on the other conveyor belt for pieces that need to be reworked.

The next sections explain how to define these rules and execute them on `MetaEdit+` models.

3. The Rule Editor

A rule editor is a visual environment for creating graph transformation rules. Graph transformation systems consist of a number of rules, which can

be executed on a model. These rules re-use the domain-specific visual notation of the elements to be transformed. They consist of three parts: exactly one left hand side (LHS), exactly one right hand side (RHS) and one or more negative application conditions (NACs). The LHS holds a pattern to indicate which part of the model that is to be matched. If the rule is executed 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 such match can be found, the rule will rewrite the model by replacing the elements in the LHS by the elements in the RHS.
The rule objects decompose into graphs conforming to the rule meta-model, which consists of the three elements mentioned above: one LHS, one RHS and one or more NACs. A rule also has a name and a precedence, hich is a positive integer. The precedence defines layers in the graph grammar. The graph grammar 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 the rules of the next layer begins. As we use model transformation for simulation, our graph grammer 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 precondition pattern metamodel. The right hand side object of a rule decomposes into a graph conforming to the postcondition pattern metamodel. We create these metamodels starting from the original metamodel and applying a process called RAMification [Kühne et al. (2010)] on them. RAM stands for Relaxation, Augmentation and Modification. . In particular, the following steps were taken to create the modified versions of the metamodel.

1. Relax the constraints on the metamodel's well-formedness. A rule often only matches a part of a model and this may not be a well-formed model conforming to the original metamodel. It's also possible for abstract superclasses to appear in rules, which is impossible in the original modelling language.
2. Append the suffix '_LHS' (precondition pattern) or '_RHS' (postcondition pattern) to the class names of the objects and relationships.
3. 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.
4. Append the suffix '_LHS' (precondition pattern) or '_RHS' (postcondition pattern) to each property of an object or a relationship and change

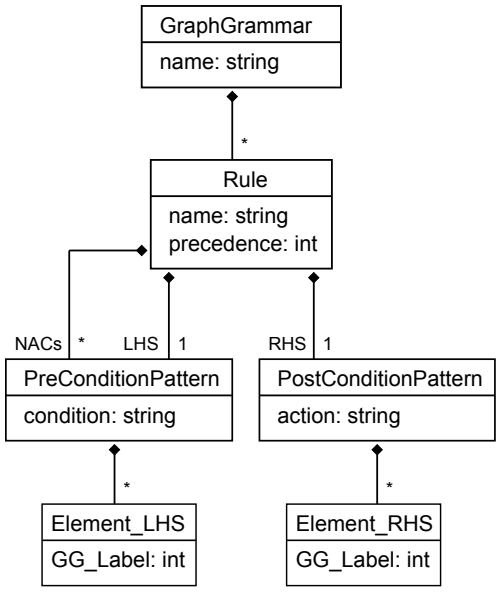

Figure 5: Structure of a graph grammar. Adapted from [Kühne et al. (2010)].

its datatype to 'String'. The properties now define a condition (precondition) or an action (postconsition pattern) instead of an actual value. The 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.

5. Add a property called 'constraint' (precondition pattern) or 'action' (postcondition 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 also has to be modified. As the layered architecture of the graph grammar gives priority to layers with a lower precedence value, mechanisms to ensure fairness have to be implemented. We have done this by adding properties to objects that are modified by the rules (see Section 2). These properties can be checked in the negative application condition(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. For instance, we've added a 'moved' property to the 'Operator' object, which has to be 0 in order for the rule 'MoveOperator' (which moves an operator from one machine to an-

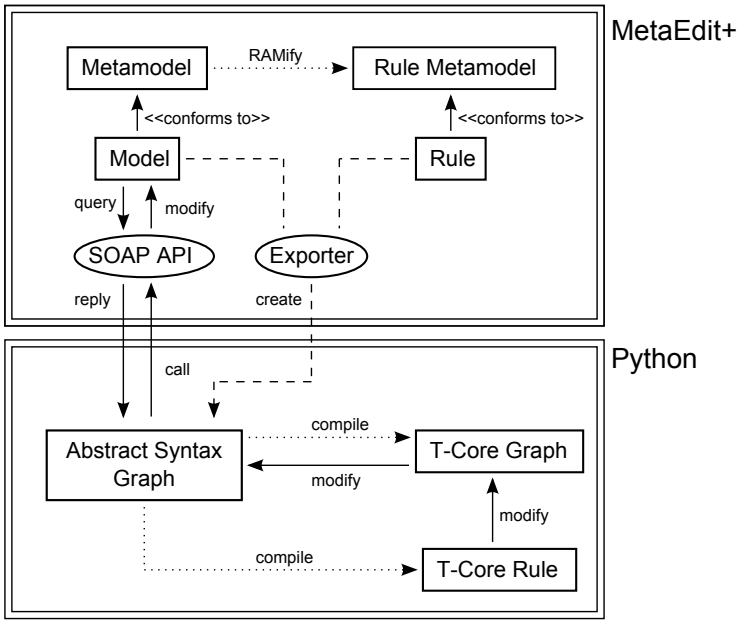

Figure 6: The architecture used for our demonstration, including calls and relations between different components.

other one) to execute. The right hand side sets this property to 1. The top layer of our graph grammar consists of rules that set these properties back to their initial values so all rules become enabled again. In that way, each pass through all the rules only considers each rule once and fairness is achieved.

4. Architecture

Our solution has a client-server architecture. The transformation engine acts as the server, `MetaEdit+` as the client. The architecture is visually represented in Figure 6.

4.1. Python: Abstract Syntax Graph

We started by creating an abstract representation of `MetaEdit+` models in Python. This component has two functions: it provides a data structure to export models to using the `MetaEdit+` generators, 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. The architecture for this ASG is visually represented in Figure 7. All necessary information is included, i.e. objects, relationships, roles and properties. Also, the area id and object id which are used by the `MetaEdit+` API to identify objects, relationships and representations are stored.

4.2. `MetaEdit+` : API and Generators

The SOAP API of `MetaEdit+` is heavily used in our demonstration. 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 3, a rule consists of exactly one LHS, one or more NACs and exactly one RHS. These components of a rule are, like models, mapped to the ASG structure in Python.

4.3. T-Core: Graph Rewriting

T-Core is a libray of graph transformation primitives. 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) will be used, which chooses one match of the set of all matches (matching the LHS, considering the NACs) and transforms the LHS to the RHS. Before we can use T-Core, we need to take care of the following.
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).
The compiler works as follows. For compiling a model, it iterates over all nodes in the ASG twice. The first time, 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 to the ASG (see Section 5). It corresponds to the area id and object id of the object, and should therefore 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

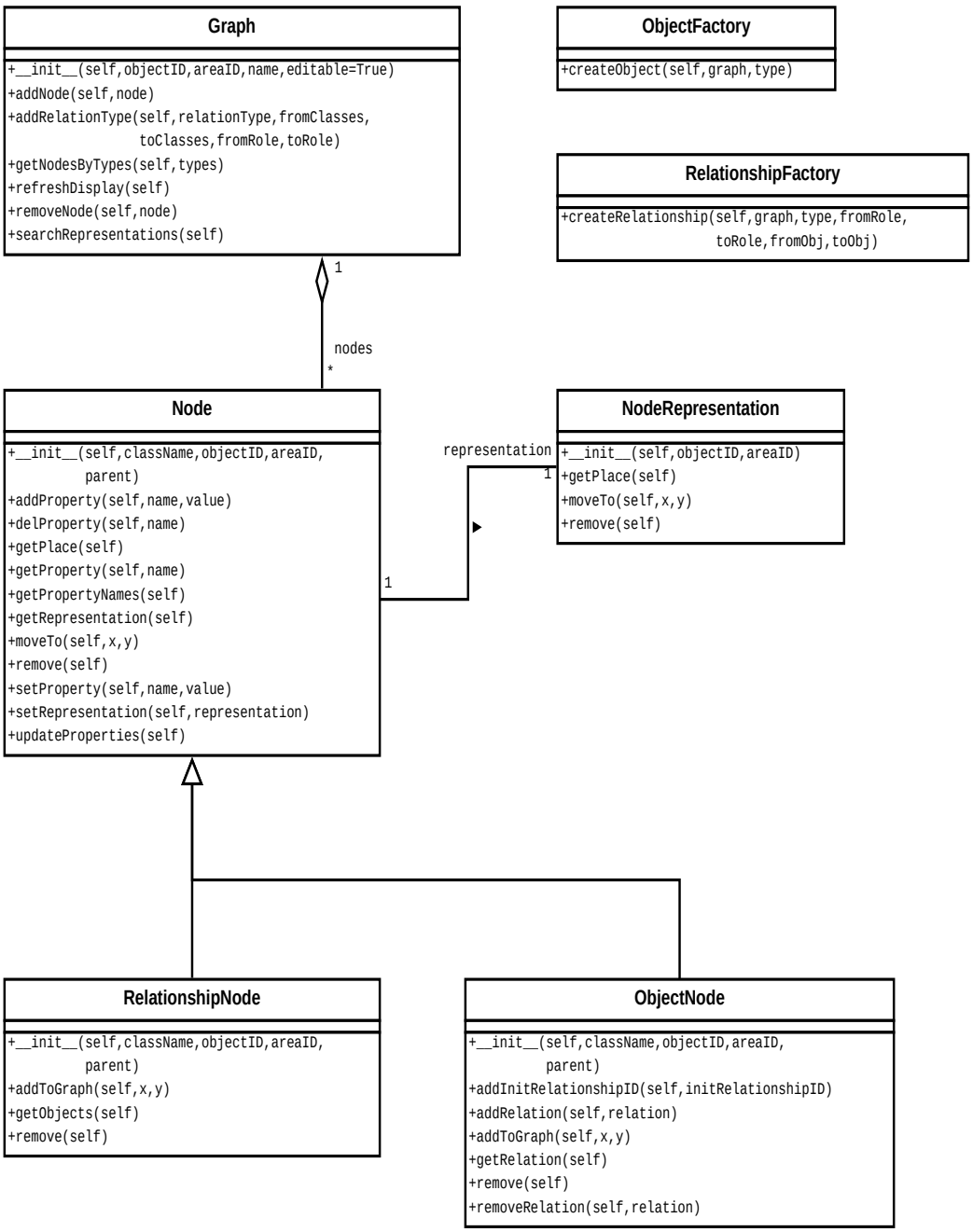

Figure 7: The architecture used for the ASG component in Python.

mechanism to achieve this by naming the property in a particular way. The second time, edges are added from relationship nodes to its source and target nodes. These aren't 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 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 string that is entered 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 in a similar way.
As both the model and rules are now represented as T-Core graphs, the rules can be executed.

5. Executing Rules

This section explains how a rule is applied on a model. We start by creating a rule in `MetaEdit+` , then exporting it to Python. There, it will be compiled together with an exported model to T-Core. T-Core will rewrite its graph and report back the changes, which will be used to modify the ASG accordingly.

5.1. Creating the Rule

We will consider the moving of an operator from one machine to another as an example rule. For a visual representation of the rule as it would appear in the rule editor, see Figure 8. An operator in our language can be connected to either an assembler, a quality control station or a repair machine. All three of these machines inherit from the processor abstract superclass. In the LHS of the rule we define what should be matched: two processors, one of which the operator is connected to. As a condition for the 'moved' property, we state that it should be equal to 0. As we do not want two operators connected to the same machine, we also define a NAC. There, the processor we want the operator to move to (with 'GG_Label' equal to 4) has an operator connected to it. By defining this NAC, we make sure 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.

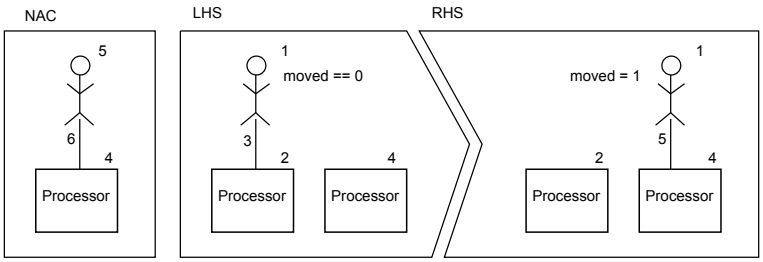

Figure 8: En example rule: the moving of an operator from one processor to another one.

There, 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.

5.2. Compiling and Executing the Rule

As was explained in Section 4.3, 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 combination of processor classes. This would lead to an explosion of the number rules. T-Core support subtype matching: a list of subtypes for each type needs to 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 in case there is more than one. 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.

5.3. Modifiying the ASG

We use the list of changes made to the T-Core graph 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 and ensures both graphs represent the same

model. On top of that, the operations that change the ASG propagate these changes through the SOAP API to the original model in `MetaEdit+` , which gives us visual feedback.

6. 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 allowed us to visually create rules, which were combined in a gaph grammar. The graph grammar was then exported to Python, where it was executed on an exported `MetaEdit+` model using T-Core as a backend. The execution of a graph grammar is a series of graph rewritings, which visually propagate to the original `MetaEdit+` model by using the SOAP API.

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 above 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 Metamodels 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.

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

7. Acknowledgments

I would like to thank several people at Metacase that have made this project possible. Steven Kelly for his continuous support and prompt replies on the MetaEdit+ forums, Juha-Pekka Tolvanen and Janne Luoma for the help they've given me. I would also like to thank Professor Hans Vangheluwe, who has assisted me on every step of this project.

References

de Lara, J., Vangheluwe, H., April 2002. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In: Kutsche, R.-D., Weber, H. (Eds.), FASE'02. Vol. 2306 of LNCS. Springer, Grenoble, France, pp. 174–188.

Helms, B., Shea, K., Hoisl, F., 2009. A framework for computational design synthesis based on graph-grammars and function-behavior-structure. ASME Conference Proceedings 2009 (49057), 841–851.
URL `http://link.aip.org/link/abstract/ASMECP/v2009/i49057/p841/s1`

Jakumeit, E., Buchwald, S., Kroll, M., 2010. Grgen.net - the expressive, convenient and fast graph rewrite system. STTT 12 (3-4), 263–271.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., June 2008. 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.

Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M., 2010. Explicit transformation modeling. In: Ghosh, S. (Ed.), Models in Software Engineering. Vol. 6002 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 240–255, 10.1007/978-3-642-12261-3_23.
URL `http://dx.doi.org/10.1007/978-3-642-12261-3_23`

Nickel, U., Niere, J., Zündorf, A., June 2000. The FUJABA environment. In: ICSE'00. ACM, Limerick (Ireland), pp. 742–745.

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