Automatic Generation of Model-to-Model Transformations from Rule-Based Specifications of Operational Semantics

Hans Vangheluwe[1] and Juan de Lara[2]

[1] School of Computer Science
McGill University (Montréal, Canada)
hv@cs.mcgill.ca
[2] Polytechnic School
Universidad Autónoma (Madrid, Spain)
jdelara@uam.es

Abstract. In this paper we present a new approach for the automatic generation of model-to-model transformations given a description of the operational semantics of the source language by means of a graph grammar. The approach is oriented to the design of transformations from Domain-Specific Visual Languages (DSVLs) into semantic domains with an explicit notion of transition, such as Petri nets. We illustrate these techniques with a DSVL in the domain of production systems, for which we generate a transformation into place/transition Petri nets starting from a rule-based description of the DSVL's operational semantics.

1 Introduction

Domain-Specific Visual Languages (DSVLs) are becoming popular in order to facilitate modelling in specialized application areas. Using DSVLs, designers are provided with high-level intuitive notations which allow building models with concepts of the domain and not of the solution space or target platform (often a low-level programming language). This makes the construction process easier, with the potential to increase quality and productivity.

Usually, the DSVL is specified by means of a meta-model describing abstract syntax concepts. The concrete syntax can be given by assigning visual representations to the different elements in the abstract syntax meta-model. For the semantics, several possibilities are available. For example, it is possible to specify semantics by using visual rules [10], which prescribe the pre-conditions for a certain action to be triggered, as well as the effects of such an action. The pre- and post-conditions are given visually as models. This technique has the advantage of being intuitive, as it uses domain concepts to describe the rules.

Graph transformation [3] is one such rule-based technique. One of the most popular formalizations of graph transformation is based on category theory [4] and supports a number of interesting analysis techniques, such as detecting rule dependencies [4, 8]. However, graph tranformation lacks advanced analysis capabilities that have been developed for other formalisms for expressing semantics, such as Petri nets [15]. This is due to the fact that (Place/Transition) Petri nets are less expressive than graph transformation.

To address the lack of analysis capabilities, another possibility for expressing the semantics of a DSVL is to specify a mapping from the source DSVL into an appropriate semantic domain. This possibility allows one to use the techniques specific to the semantic domain for analyzing the source models. However, this approach is sometimes complicated and requires from the DSVL designer deep knowledge of the target language in order to specify the transformation.

To reap the benefits of both approaches, we have developed a technique for deriving a transformation from the source DSVL into a semantic domain, starting from a rule-based specification of the DSVL semantics (graph transformation in particular [3]). Such a specification uses domain-specific concepts only and is hence domain-specific in its own right. In addition, such behavioural specification may include control structures for rule execution (such as layers [1] or priorities [10]). The main idea is to automatically generate triple graph grammar (TGG) rules [16]

to first transform the static information (i.e., the initial model) and then the dynamics (i.e., the rules expressing the behaviour and the rule control structure). We exemplify this technique by using (Place/Transition) Petri nets [15] as the target language. Other formalisms with an explicit representation of a "simulation step" or transition (such as Constraint Multiset Grammars [13] and process algebras) could also be used. This explicit representation of a transition allows encoding the rule dynamics in the target model.

Paper organization. Section 2 presents the rule-based approach for specification of behaviour, together with the example that will be used throughout the paper: a production system. Section 3 shows how the initial model (i.e., the static information) is transformed to the Petri net semantic domain. Section 4 presents the approach for translating the rules and the control structure to the Petri net semantic doamin. Section 5 presents related research and finally, Section 6 ends with the conclusions. Due to space limitations we keep the discussion at an informal level, omitting a theoretical presentation of the concepts.

2 Rule-Based Specification of Operational Semantics

In this section we provide a description of a DSVL for production systems using meta-modelling, and its operational semantics using graph transformation. Note that the latter will be described using concrete syntax, thus making the expression of operational semantics domain-specific. Figure 1 shows in the upper part a meta-model for the example language. The language contains different kinds of machines, which can be connected through conveyors. Human operators are needed to operate the machines, which consume and produce different types of pieces from/to conveyors. Conveyors can also be connected.

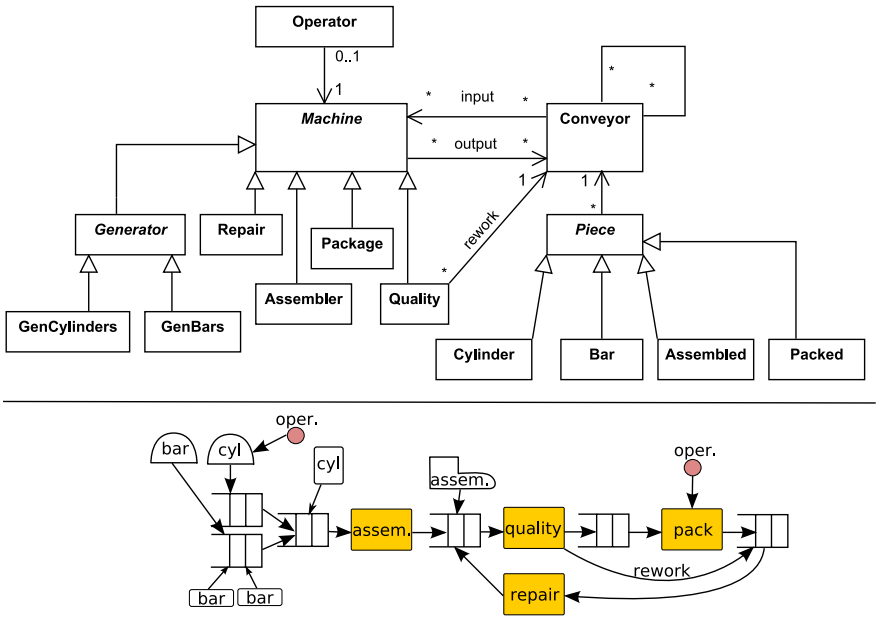

Fig. 1. Meta-Model for the Example Language (up). Example Model (down)

The lower part of Figure 1 shows a production model example using a concrete visual syntax. It contains six machines (one of each type), two operators, six conveyors and four pieces. Machines are represented as boxes, except generators, which are depicted as semi-circles with the kind of piece they generate inside. Operators are shown as circles, conveyors as lattice boxes, and each kind of piece has its own shape. In the model, the two operators are currently operating a generator of cylindrical pieces and a packaging machine respectively.

Figure 2 shows some of the graph transformation rules that describe the DSVL's operational semantics. Rule "assemble" specifies the behaviour of an assembler machine, which converts one

cylinder and one bar into an assembled piece. The rule can be applied if every specified element (except those marked as "{new}") can be found in the model. When such an occurrence is found, then the elements marked as "{del}" are deleted, and the elements marked as "{new}" are created. Rule "change" models the fact that an operator may move from one machine (of any kind) to another one when the target machine is unattended and it has at least one incoming piece (of any kind). The rule has a negative application condition (labeled as *NAC*), which forbids its application if the target machine is already being controlled by an operator. Note that in this case we use *abstract objects* in rules (e.g., Machine is an abstract class). Of course, no object with an abstract typing can be found in the models, but the abstract object in the rule can get instantiated to objects of any concrete subclass [11] (this not only applies to abstract classes, but to any class having subclasses). This way, multiple rules can be lumped into one. The rule in the example is equivalent to 36 rules, resulting from the substitution of each *machine* by its children concrete classes.

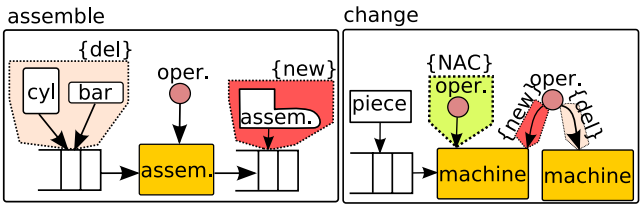

Fig. 2. Some Rules for the Production Systems Language.

By default, graph grammars use a non-deterministic execution model. This, in order to perform a *direct derivation* (i.e., a simulation step), a rule is chosen at random, and is applied if its pre-condition holds in some area of the model. This is a second source of non-determinism, as a rule may be applicable in different parts of the model, and then one is chosen at random. The grammar execution ends when no more rules are applicable. Different rule control structures can be imposed on the grammar to reduce the first source of non-determinism, and to make the transformation models more readable and re-usable. We will present two types of control structures (layers and priorities) later in section 4.1.

As the example has shown, graph transformation is an intuitive means to describe the operational semantics of a DSVL. Its analysis techniques are limited however. It is for example difficult to determine termination and confluence (which for the general case are non-decidable), state reachability, reversibility, conservation and invariants. To make analysis possible, the next sections show how to automatically obtain a transformation into place/transition Petri nets starting from the previous rule-based specification.

3 Transforming the Static Information

In this section, we explain how, starting from the previous definition of the syntax and semantics of the Production Systems DSVL, a transformation into Petri nets can be automatically derived. In a first step, the static information of the source model is transformed. For this purpose, the designer has to select the Petri net roles that the elements of the source language will play in the target language. This is specified with a *meta-model triple* [7], which is a structure specifying the allowed relations between two meta-models. A meta-model triple for the example is shown in Figure 3.

This process of identifying roles for source elements can be understood as a kind of *model marking* [14], i.e., annotating the model before the transformation takes place. In the example, we state that machines and conveyors play the roles of *places* in Petri nets (i.e., they are holder-like elements), whereas operators and pieces are *token*-like entities (i.e., they can "move around",

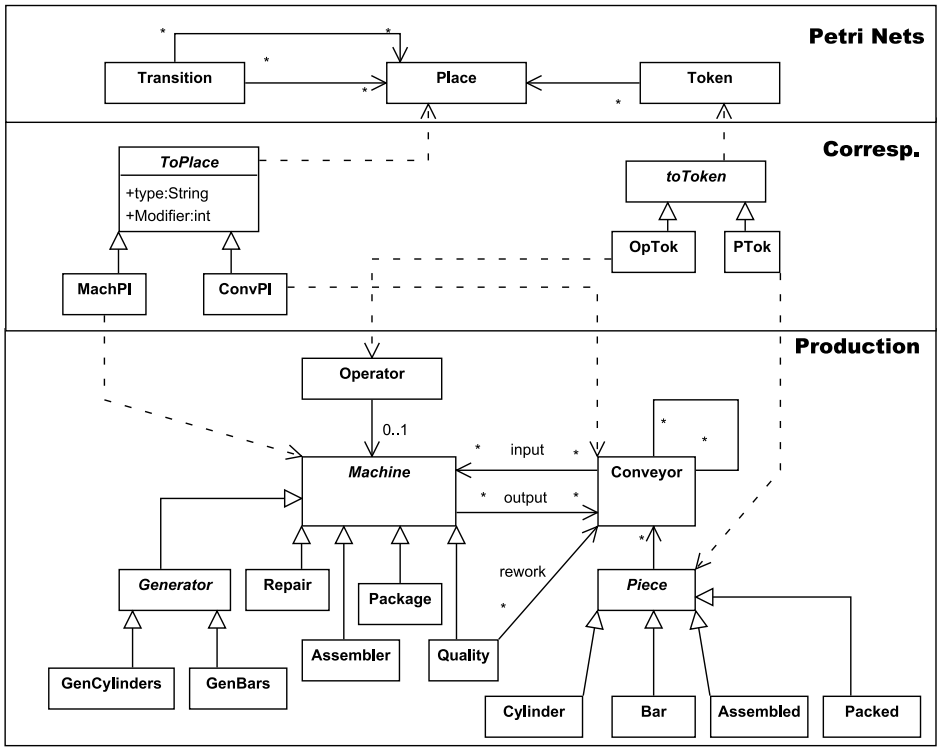

Fig. 3. Meta-Model Triple for the Transformation.

being associated with machines and conveyors respectively). For this particular transformation into Place/Transition Petri nets, the meta-model triple provides two standard mappings: *ToPlace* and *ToToken*, which allow relating source elements to places and tokens respectively, by subclassing both classes. Please note that, as we are translating the *static* information, no element can play the role of a Petri net transition. As the next section will show, the role of Petri net transitions is reserved for the dynamic elements in the source specification: the rules modelling the operational semantics.

From the meta-model triple, a number of *operational* TGG rules [16] are generated. These rules manipulate structures (triple models) made of source and target models, and their interrelations. The rules specify how the target model should be modified taking into consideration the structure of the source model. Thus, TGG rules manipulate triple models conforming to a meta-model triple (such as the one in Figure 3).

The TGG rules we automatically generate associate with each place-like entity (in the source language) as many places as there are different types of token-like entities connected to it in the meta-model. In the example, class *Machine* (place-like) is connected to class *Operator*, a token-like entity. Thus, we have to create one place for each machine in the model. Conveyors are also place-like, and are connected to pieces (token-like). Thus, we have to create four different places for each conveyor (to store each different kind of piece). This is necessary as tokens are indistinguishable (as we are not using coloured Petri nets). Distinguishing them is done by placing them in distinct Petri net places.

Figure 4 shows some of the resulting TGG rules. For example, rule "add 1-op-machine" associates a place with each machine in the source model (due to the fact that operators can be connected to machines). The place in the target model, together with the mapping to the source element is marked as *new* (so it is created), and also as *NAC*, so that it can only be created once for each source machine. Rule "init 1-op-machine" creates the initial marking of places associated with machines. It adds one token in the place associated with each machine for every operator connected to it. We represent tokens as black dots connected to places.

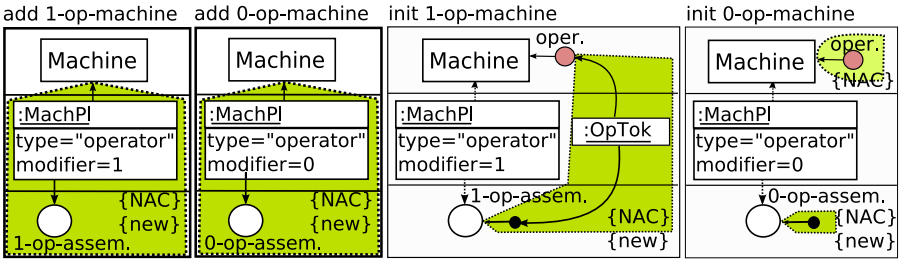

Fig. 4. Some TGG Rules for Translating the Model.

In addition, as the number of operators in each machine is bounded (there is a "0..1" cardinality in the source meta-model), an additional place (that we call *zero-testing* place) is associated with machines in order to represent the absence of operators in the given machine. This is performed by the automatically generated rule "add 0-op-machine". The initialization of this zero-testing place is done by rule "init 0-op-machine", which adds a token in the place if no operator is connected to the machine. These places will be used later on to test negative conditions. Note that we cannot generate such zero-testing places for conveyors, as the number of pieces that can be stored in a conveyor is not bounded. This restricts the kind of negative tests that can be done for conveyors. Note that these zero-testing places are not needed if the target language has built-in primitives for this kind of testing, such as Petri nets with inhibitor arcs [15]. These kinds of nets, though more expressive, allow fewer analyses however. Reachability for example is not decidable in a net with at least two inhibitor arcs.

Applying the generated rules to the source model in Figure 1, the Petri net in Figure 5 is obtained (we do not show the mappings to the source model for simplicity). The next section shows how the translation of the operational semantics (dynamics) of the production systems DSVL is performed.

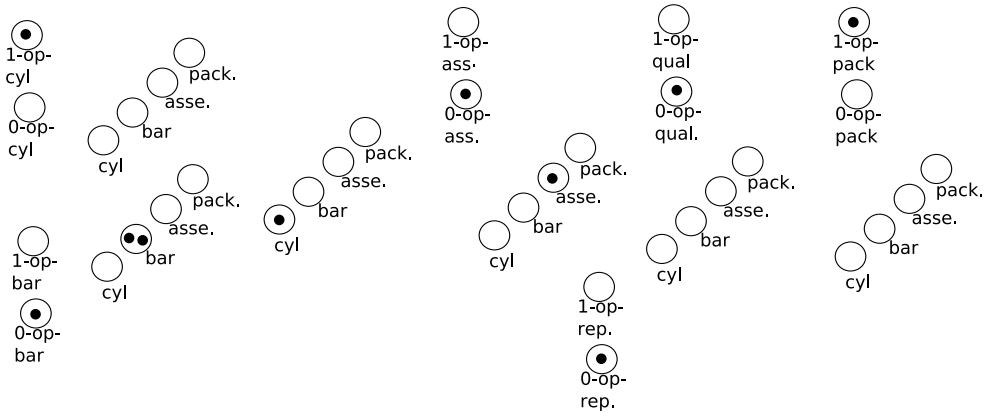

Fig. 5. First Step in the Transformation.

4 Transforming the Dynamic Behaviour

In order to translate the rules (shown in Figure 2) describing the operational semantics of the production systems DSVL into the Petri net target language, a number of additional TGG rules are generated. These rules "embed" each operational rule in the target language. Thus, in our case, we make explicit in the Petri net (by means of transitions) all allowed movements of token-like entities: pieces and operators. This reflects the fact that rules for the movement of pieces and operators in the source language can be applied non-deterministically at each possible occurrence.

Figure 6 shows one of the generated rules. Rule "create change" is created from rule "change" in Figure 2, and adds a Petri net transition to model the movement of operators between machines. The transition checks that a token is present in the place corresponding to the conveyor. The firing of the transition creates a token in the place corresponding to the machine connected to the conveyor, and deletes a token from the place corresponding to the other machine. The NAC in rule "change" has been translated by using the zero-testing place associated with the target machine (to ensure that it is currently unattended). Note however that the original rule "change" cannot have NACs involving pieces, as we may have an unbounded number of these in conveyors. This indicates that our approach will not be able to translate all possible graph transformation rules, due to the limited expressiveness of place/transition Petri nets.

The TGG rule uses the source model to identify all relevant place-like elements in the pre- and post- conditions of the original rule. It should be noted that this TGG rule will be applied at each possible occurrence of two machines where one has an incoming conveyor, producing a corresponding Petri net transition in the target model. Moreover, as the original rule does not specify the kind of piece in the conveyor, a transition is created for each different type (note the "type=x" attribute). Thus, we are identifying a priori (by adding Petri net transitions) *all possible instantiations* (in the production system model) of the rules implementing the operational semantics.

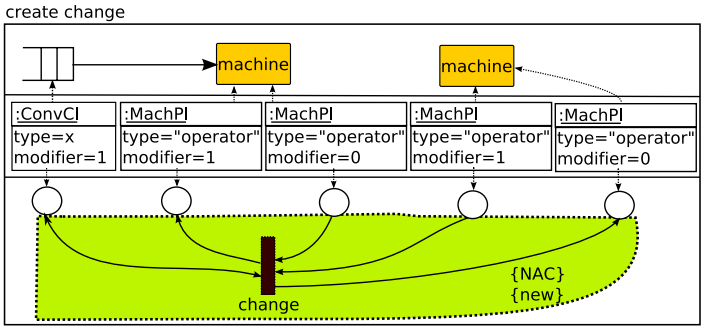

Fig. 6. One of the TGG Rules for Translating the Operational Rules.

Figure 7 shows the result of applying the generated triple rules to the model in Figure 5. It is only partially shown for clarity, as many other transitions would be generated. In particular, we show only two instantiations of rule "change" to move an operator to another machine: from the assembler machine to quality checking the availability of assembled pieces (transition labelled *a2q-ass*) and from quality to package seeking assembled pieces (transition labelled *q2p-ass*). The full transformation generates transitions to move the operators between all combinations of machines and piece types.

4.1 Transforming the Execution Control for the Rules

Up to now, we have not assumed any control structure for rule execution. That is, rules are tried at random, and the execution finishes when no more rules are applicable. With this control scheme, no further transformations are needed, and in the example the resulting Petri net is the one in Figure 7.

However, it is also possible to translate rule control structures, explicitly modellling semantics of programmed graph transformation schemes. For example, one can assign priorities to rules [10], in such a way that rules with higher priorities are executed first. If more than one rule has the same priority, one is executed at random. All rules with the highest priority are executed as long as possible. Each time such a rule is executed, the control goes back to the rules with the highest priority. When no more rules with the highest priority can be executed, the control goes to rules

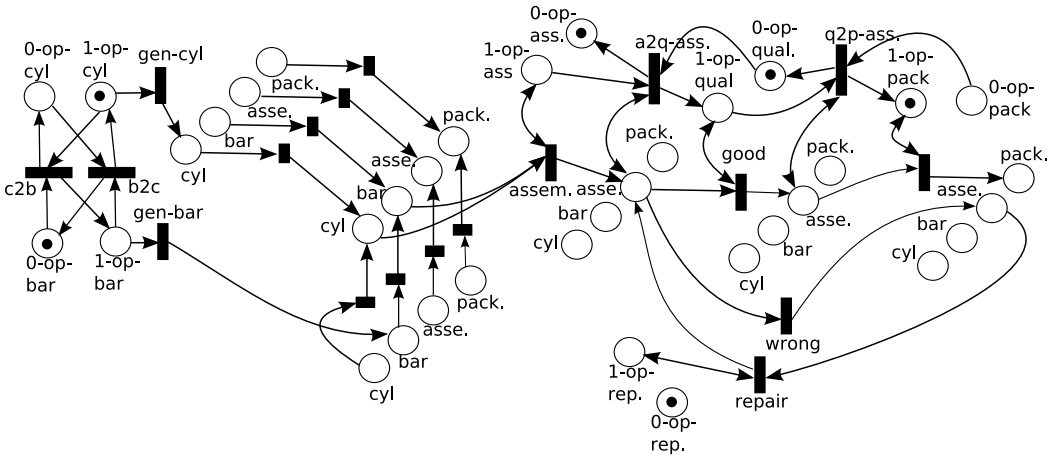

Fig. 7. Second Step in the Transformation (some transitions omitted for clarity).

with the next lower priority, and so on. The execution ends when none of the rules with the lowest priority can be executed.

This execution policy can be embedded in the resulting Petri net as well. We illustrate the translation with the scheme shown in Figure 8. The figure assumes two rules ($r1$ and $r2$) with the highest priority (priority one). In addition, these transitions would be connected to the pre- and post- condition places, resulting from the previous step in the transformation (described in the previous section). The idea is that in the priority 1 phase, modelled by the *prio-1* place, rules $r1$ and $r2$ are tried. Both cannot be executed, because place $p1+$ makes them mutually exclusive. Transitions $\neg r1$ and $\neg r2$ are constructed from the operational rule specifications in such a way that they can be fired whenever $r1$ and $r2$ cannot be fired, respectively (details will be shown later). Thus, if both $\neg r1$ and $\neg r2$ are fired, the control goes to the next priority (as this means that $r1$ nor $r2$ can be executed). If either $r1$ or $r2$ can be fired, then the control remains in priority one. The transitions that move the priority token take care of removing the intermediate tokens from $r1$, $r2$, $\neg r1ex$ and $\neg r2ex$.

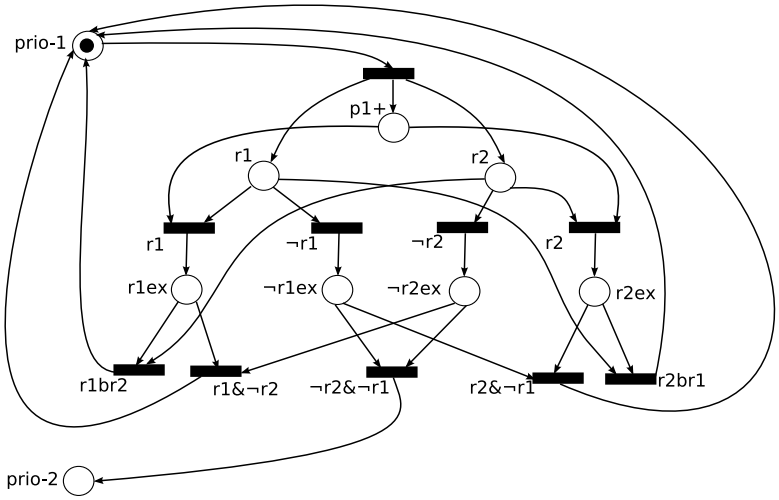

Fig. 8. Scheme for Transforming a Control Structure Based on Priorities.

Thus, an important issue in this transformation is that we need to check when rules are *not* applicable (as transitions $\neg r1$ and $\neg r2$ did in the previous figure). This in general is only possible if the places associated with the rule are bounded. Thus, in the case of the example of previous sections, we can*not* test whether rules "assemble" or "change" cannot be fired, since the number of pieces in conveyors is not bounded.

Figure 9 shows an example of the construction of the transitions for testing non-executability of rule "rest", which models the deletion of an operator. Triple rule "create ¬rest" generates a Petri net transition that tests if the machine is not attended. If this is the case, transition "¬rest" can fire, which means that "rest" cannot (i.e., the rule cannot be applied at that match).

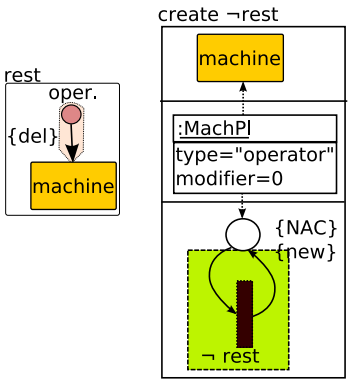

Fig. 9. Example of Generation of Rule for Testing Non-Applicability.

Typical control structures in graph transformation, such as layers can be transformed in a similar way as priorities. Note that the transformation of the control structure can be kept independent of the two previous transformation steps (transforming the static structure and the operational rules). We are thus in effect *weaving* two transformations.

5 Discussion and Comparison with Related Work

Many contributions to model-to-model transformation have concentrated on devising high-level means to express them. On the more formal side, we can find for example the seminal work on TGGs [16], which proposed an algorithm to generate operational rules (performing for example source-to-target or target-to-source translations) from declarative ones. Recent works try to provide even higher-level means to express the transformations, using for example *triple patterns* [12] from which operational TGG rules are generated. This is closely related to the notion of "model transformation by example" [17, 19] (where transformation rules are derived starting from a mapping between two meta-models) and transformation models [2] (which express transformations as a MOF model relating source and target elements, and OCL constraints).

Note however that our work is intrinsically different from these, as what we really do is to express the semantics of the graph grammar rules (that express the operational semantics of the source model) into Petri nets. The resulting Petri net can be seen as a restricted kind of graph grammar, as the token game (the Petri net semantics) can be considered as a graph transformation step on discrete graphs [5]. Some related work has tried to encode graph transformation rules in Petri nets, and then use the analysis techniques of the later to investigate the former. For example, in [18] a graph transformation system is abstracted into a Petri net to study termination. However, there are three fundamental differences with our work:

- They only consider rules, while we consider rules and an initial graph, therefore we are able to consider all possible instantiations (occurrences) of the source rules.
- They end up with an abstraction of the original semantics, as, when the transformation is done, the topology of the source model is lost (i.e., tokens represent instances of the original types, but their connections are lost). However, the fact that we consider an initial model allows us to retain its topology, thus the transformation does not lose information (the obtained Petri net perfectly reflects the semantics of the original language). This is given by the fact that a Petri net transition is generated for each possible application of the original rule.

– We consider control structures for the rules. In fact, our approach allows for experimenting with different control structures by explicitly mapping them onto Petri net structures.

6 Conclusions

In this work we have presented a new technique for the automatic generation of a model-to-model transformation (into a semantic domain) given a rule-based specification of the operational semantics of the source language. The presented technique has the advantage that the language designer almost exclusively needs to work with the concepts of the source DSVL, and does not have to provide the model-to-model transformation nor have knowledge in the target formalism.

We have illustrated this technique by transforming a production system into a Petri net. The designer has to specify the simulation rules for the source language, and the roles of the source language elements. From this information, a number of TGG rules are generated that perform the transformation. Once the transformation is performed, the Petri net can be simulated or analyzed, for example to check for deadlocks or state reachability. Thus, by using the Petri net techniques, we can answer difficult questions about the original operational rules, such as termination or confluence (which for the case of general graph grammars are undecidable – note however that we cannot work with all possible graph grammars).

In the future, we will work on full tool support (using our tool for multi-formalism and meta-modelling AToM^3 [10]) for this transformation generation, as well as study other target languages. Note that tool support requires the ability to model and transform model transformations (i.e., higher-order transformation). It can also be interesting to analyze how to translate other more advanced control structures, like transformation units [9] or story diagrams [6].

Acknowledgements. We'd like to thank the referees for their useful comments. Juan de Lara's work has been partially sponsored by the Spanish Ministry of Science and Education, project MOSAIC (TSI2005-08225-C07-06).

References

1. AGG, The Attributed Graph Grammar System, home page at: `http://tfs.cs.tu-berlin.de/agg/`.
2. Bezivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A. 2006. *Model Transformations? Transformation Models!*. Proc. MoDELS'06, LNCS 4199, pp.: 440-453, Springer.
3. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1.* World Scientific.
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.
5. Ehrig, H., Padberg, J. 2004. *Graph Grammars and Petri Net Transformations.* Proc. ACPN'2003, LNCS 3098, pp.: 496-536, Springer.
6. Fischer, T., Niere, J., Torunski, L., Zündorf, A. 1998. *Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java.* Proc. 6th Int. Workshop on Theory and Application of Graph Transformations. LNCS 1764, pp.: 296 - 309. Springer.
7. Guerra, E., de Lara, J. 2007. *Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages.* Software and Systems Modeling 6(3), pp.: 317-347. Springer.
8. Heckel, R., Küster, J. M., Taentzer, G. 2002. *Confluence of Typed Attributed Graph Transformation Systems.* Proc. ICGT'02, LNCS 2505, pp.: 161-176, Springer.
9. Kreowski, H.-J., Kuske, S. 1999. *Graph transformation units and modules.* In Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools, pp.: 607 - 638. World Scientific.
10. de Lara, J., Vangheluwe, H. 2004. *Defining Visual Notations and Their Manipulation Through Meta-Modelling and Graph Transformation.* JVLC, Vol 15(3-4), pp.: 309-330. Elsevier.

11. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer G. 2007. *Attributed graph transformation with node type inheritance.* Theor. Comput. Sci. 376(3), pp.: 139-163.
12. de Lara, J., Guerra, E., Bottoni, P. 2007. *Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules.* Proc. GT-VMT' 2007. Electronic Communications of the EASST, Vol 6.
13. Marriott, K., Meyer, B., Wittenburg, K. 1998. *A survey of visual language specification and recognition.* Theory of Visual Languages. Pages 5-85. Springer-Verlag.
14. Mellor, S., Scott, K., Uhl, A., Weise, D. 2004. *MDA Distilled: Principles of Model-Driven Architecture.* Addison Wesley.
15. Peterson, J. L. 1981. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, N.J.
16. Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science. LNCS 903, pp.: 151 - 163. Springer.
17. Varro, D. 2006. *Model Transformation by Example.* Proc. MoDELS'06, LNCS 4199, pp.: 410-424, Springer.
18. Varro, D., Varro - Gyapay, S., Ehrig, H., Prange, U., Taentzer, G. 2006. *Termination Analysis of Model Transformations by Petri Nets.* Proc. ICGT'06, LNCS 4178, pp.: 260-274, Springer.
19. Wimmer, M., Strommer, M., Kargl, H., Kramler, G. 2007. *Towards Model Transformation Generation by Example.* Proc. 40th Hawaii International Conference on System Sciences, HICSS'07, pp.: 285, IEEE Comp. Society.