Translating Model Simulators to Analysis Models

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

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

Abstract. We present a novel approach for the automatic generation of model-to-model transformations given a description of the operational semantics of the source language by means of graph transformation rules. The approach is geared to the generation of transformations from Domain-Specific Visual Languages (DSVLs) into semantic domains with an explicit notion of transition, like for example Petri nets. The generated transformation is expressed in the form of operational triple graph grammar rules that transform the static information (initial model) and the dynamics (source rules and their execution control structure). We illustrate these techniques with a DSVL in the domain of production systems, for which we generate a transformation into Petri nets.

1 Introduction

Domain-Specific Visual Languages (DSVLs) are becoming increasingly popular in order to facilitate modelling in specialized application areas. Their use in software engineering is promoted by recent development paradigms such as Model Driven Development (MDD). 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, having the potential to increase quality and productivity.

Usually, the DSVL is specified by means of a meta-model with the abstract syntax concepts. Additionally, the concrete syntax can be given by assigning visual representations to the different elements in the meta-model. For the semantics, several possibilities are available. For example, it is possible to specify semantics by using visual rules [4,8], which describe the pre-conditions for a certain action to be triggered, as well as the effects of such action. The pre- and post- conditions are given visually as models that use the concrete syntax of the DSVL. This technique has the advantage of being intuitive, as it uses concepts of the domain for describing the rules, thus facilitating the specification of simulators for the given DSVL.

Graph transformation [3] is one such rule-based technique. One of the most commonly used formalizations of graph transformation is based on category

J. Fiadeiro and P. Inverardi (Eds.): FASE 2008, LNCS 4961, pp. 77–92, 2008.
© Springer-Verlag Berlin Heidelberg 2008

theory [4] and supports a number of interesting analysis techniques, such as detecting rule dependencies [1,4,7]. However, graph transformation lacks advanced analysis capabilities that have been developed for other formalisms for expressing semantics, such as Place/Transition Petri nets (P/T nets) [14]. In this case, the high-power analysis is thanks to the fact that P/T nets are less expressive than graph transformation.

To address the lack of analysis capabilities, another common technique for expressing the semantics of a DSVL is to specify a mapping from the source DSVL into a semantic domain [7,8] and then back-annotate the analysis results to the source notation. This possibility allows one to use the techniques specific to the semantic domain for analysing 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 using graph transformation [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 [8]). The main idea is to automatically generate triple graph grammar (TGG) rules [15] 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 P/T nets as the target language, but other formalisms with an explicit representation of a "simulation step" or transition (such as Constraint Multiset Grammars [12] and process algebras) could also be used. This explicit representation of a transition allows encoding the rule dynamics in the target model by creating a transition for each possible execution (i.e., match) of the original rule.

Paper organization. Section 2 presents the rule-based approach for specification of behaviour by means of a DSVL for production systems. Section 3 shows how the initial model (i.e., the static information) is transformed. Section 4 presents the approach for translating the rules and the control structure. Section 5 gives an overview of the algorithms for the generation of the TGG rules. Section 6 presents related research and finally, Section 7 ends with the conclusions. Due to space limitation we keep the discussion at an informal level, omitting a theoretical presentation of the concepts when possible. A short, preliminary version of some parts of this work appeared as a technical report [16].

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. The top of Fig. 1 shows a meta-model for the example language. It contains different kinds of machines (all concrete subclasses of Machine), which can be connected through conveyors. Human operators are needed to operate the

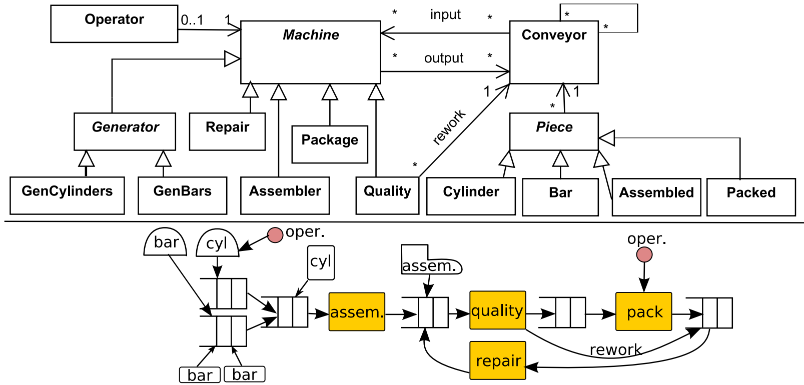

Fig. 1. Meta-Model for the Example Language (up). Example Model (down).

machines, which consume and produce different types of pieces from/to conveyors. These machines can be connected.

The bottom of Fig. 1 shows a production model example using a visual concrete 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.

Fig. 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 a 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. Note that even if we depict rules using this compact notation, we use the Double Pushout (DPO) formalization [4] in our graph transformation rules. In practice, this means that a rule cannot be applied if it deletes a node but not all its adjacent edges. In addition, we consider only injective matches.

Rule "move" describes the movement of pieces through conveyors. The rule has a negative application condition (NAC) that forbids the movement of the piece if the source conveyor is also connected to any kind of machine having an operator. In this case we use *abstract objects* in rules (i.e., piece and machine are abstract classes). 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 [9]. In this way, rules become much more compact. The rule in the example is equivalent to 24 concrete rules, resulting from the substitution of *piece* and *machine* by their children concrete classes.

Finally, 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 NAC forbids its application if the target machine is already being controlled by an operator. This rule is also abstract and equivalent to 144 concrete rules. Additional rules, not shown in the paper, model the behaviour of the other machine types.

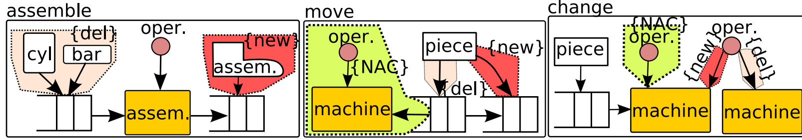

Fig. 2. Some Rules for the Production Systems DSVL

By default, graph grammars use a non-deterministic execution model. In this way, 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 match is chosen at random. The grammar execution ends when no more rules are applicable. Different rule control structures can be imposed on grammars to reduce the first source of non-determinism, and to make them more usable for practical applications. We present two of them (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, as is for example difficult to determine termination and confluence (which for the general case are non-decidable), state reachability, reversibility, conservation and invariants. For these purposes, the next sections show how to automatically obtain a transformation into P/T nets starting from the previous rule-based specification (with rules using the DSVL syntax).

3 Transforming the Static Information

In this and the next sections, we explain how, starting from the previous definition of the DSVL syntax and semantics, a transformation into P/T nets can be automatically derived. We illustrate the techniques by example, the details of the constructions are left to Section 5.

In a first step, the static information of the source model is transformed. For this purpose, the designer has to select the roles that the elements of the source DSVL will play in the target language. This is specified with a *meta-model triple* [6], a structure declaring the allowed relations between two meta-models. A meta-model triple for the example is shown in Fig. 3. The Petri nets meta-model is in the lower component, the meta-model of the source DSVL is placed in the upper component, while the correspondence meta-model in the middle is used to relate elements of both meta-models. The references (dotted arrows) depict the allowed relations for the elements in the other two meta-models. These

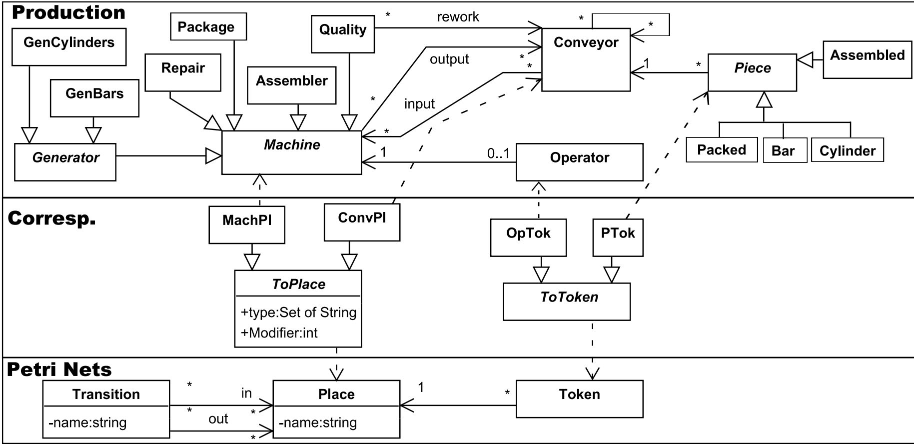

Fig. 3. Meta-Model Triple for the Transformation

references are inherited, thus, for example a "Repair" object can be related to a "Place" object through a mapping object of type "MachPl".

This process of identifying roles for source elements is a kind of *model marking* [13], i.e., annotating the model before the transformation actually 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 or place-like elements), whereas operators and pieces are *token*-like entities (i.e., they can "move around", being associated with machines and conveyors respectively). For this particular transformation into P/T 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. 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 transition is reserved for the dynamic elements in the source specification: the rules modelling the operational semantics.

From this meta-model triple, a number of *operational* TGG rules [15] are generated. These rules manipulate structures (triple models) made of source and target models, and their interrelations. They specify how the target model (a Petri net in our case) 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 Fig. 3).

The TGG rules we automatically generate associate with each place-like entity (in the source language) as many places as different types of token-like entities are 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 in P/T nets. Distinguishing them is done by placing them

in distinct places. We give additional details of this construction in Section 5. Here we only give some insight through examples.

Fig. 4 shows some of the resulting TGG rules. Rule “add 1-Op-Machine” associates a place to each machine in the source model (because 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 is created only once for each source machine. Attribute “type” of the mapping object stores the type (and all supertypes) of the token-like entity associated with the place. Rule “init 1-Op-Machine” creates the initial marking of the places associated to machines. It adds one token in the place associated to each machine for every operator connected to it. We represent tokens as black dots connected to places. Rule “add 1-Cyl-Conv” associates one place (of type “cylinder”) to every conveyor in the source model. Similar rules associate additional places for each concrete type of piece in the source meta-model.

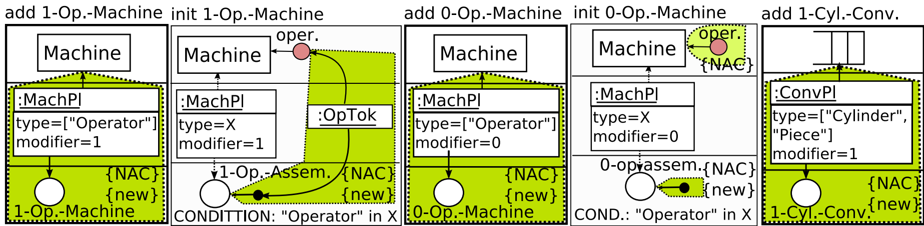

Fig. 4. Some TGG Rules for Transforming the Model

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 (which we call *zero-testing* place) is associated to machines to denote the absence of operators in the given machine. This is performed by the automatically generated rule “add 0-Op-Machine”. Distinguishing between normal places and zero-testing ones is done through the *modifier* attribute of the mapping object. The initialization of the zero-testing place for operators is done by rule “init 0-Op-Machine”, which adds a token in the place if no operator is connected to the machine. We use this kind of places to test negative conditions on token-like entities (e.g., NACs as well as non-applicability of rules). We cannot generate such kinds of 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. The zero-testing places are not needed if the target language has built-in primitives for this kind of testing, like Petri nets with inhibitor arcs [14]. These kinds of nets, though more expressive, have fewer analysis capabilities. Reachability for example is not decidable in a net with at least two inhibitor arcs.

Applying the generated rules to the source model in Fig. 1, the Petri net in Fig. 5 is obtained (we do not show the mappings to the source model for simplicity, but tag each group of places with the type of the source holder-like element). The next section shows how the translation of the dynamics is performed.

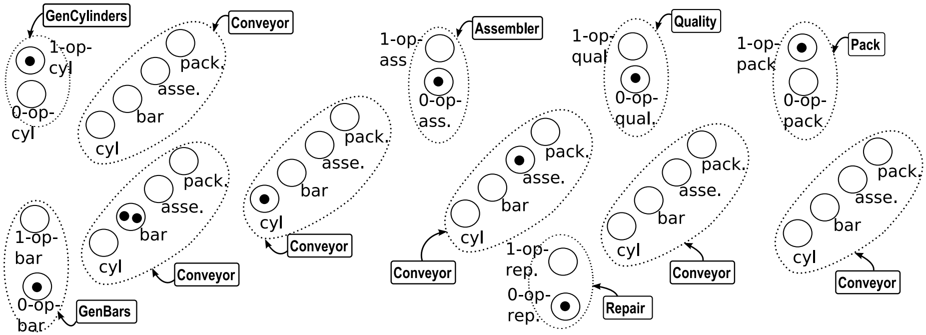

Fig. 5. First Step in the Transformation

4 Transforming the Dynamic Behaviour

In order to translate the rules implementing the operational semantics (shown in Fig. 2) into the target language, a number of additional TGG rules are needed. These rules "embed" each operational rule in the target language, in each possible way (i.e., for each possible match of the original rules in the initial model). Thus, in our case, we make explicit in the Petri net (by means of transitions) all allowed movement 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.

Fig. 6 shows some of the generated rules. Rule "create assemble" is generated from rule "assemble" in Fig. 2. It creates a Petri net transition that takes two pieces (a cylinder and a bar), checks that an operator is present, and then generates an assembled piece. The triple rule uses the source model to identify all relevant place-like elements in the pre- and post- conditions of the operational rule. This TGG rule will be applied at each possible occurrence of two conveyors connected by an assembler machine, producing a corresponding Petri net transition in the target model. Thus, we are identifying a priori (by adding Petri net transitions) all possible instantiations of the rules implementing the operational semantics. This can be done because the TGG rules contain as pre-conditions the place-like entities present in the pre-conditions of the original rules.

Rule "create change" is generated from rule "change" in Fig. 2, and adds a Petri net transition to model the movement of operators between any two machines. 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 them in conveyors. Moreover, we allow an arbitrary number of NACs in the original rules, but each one of them is restricted to have at most one token-like element, as otherwise we cannot test such condition in the Petri net in one step.

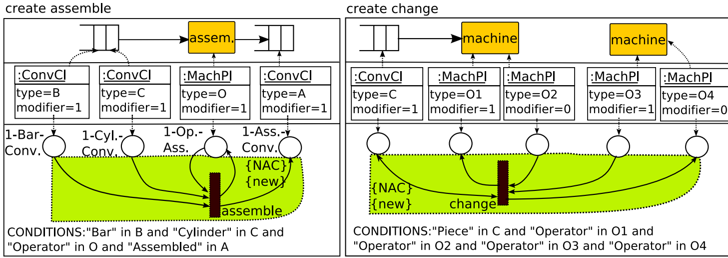

Fig. 6. Some TGG Rules for Translating the Operational Rules

Fig. 7 shows the rules generated from rule "move" in Fig. 2. Note that the original rule has a NAC involving both token-like and place-like entities. TGG rule "create move-1" assumes that the place-like entities exist, and therefore the token-like entities must not exist. The latter condition is tested by means of the zero-testing place. TGG rule "create move-2" assumes that the place-like entities do not exist. As can be seen in the rules, the handling of abstract objects in the original rule depends on their role. On the one hand, the abstract place-like entites are copied in the TGG rule (e.g., machine in the rule). On the other hand, abstract token-like elements (e.g., piece element in the rule) are handled by the attribute "type" of the mapping object (this also ocurred in rule *change*).

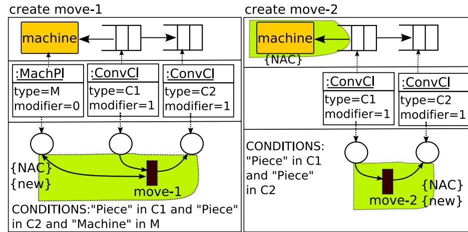

Fig. 7. Additional TGG Rules for Translating the Operational Rules

Fig. 8 shows the result of applying the generated triple rules to the model in Fig. 5. It is only partially shown for clarity, as many other transitions are generated. In particular, we show only two applications of rule "change" to move an operator to another machine: from the assembler machine to quality checking the availability of an assembled piece (transition labelled *a2q-ass*) and from quality to package seeking an assembled piece (transition labelled *q2p-ass*). The full transformation generates transitions to move the operators between all combinations of machines and types of pieces. Transitions "c2b" and "b2c" are generated by a specialized rule "change generator" (not shown) applicable to generator machines (which do not need an incoming conveyor). Again, the mappings from the Petri net places to the original model are omitted.

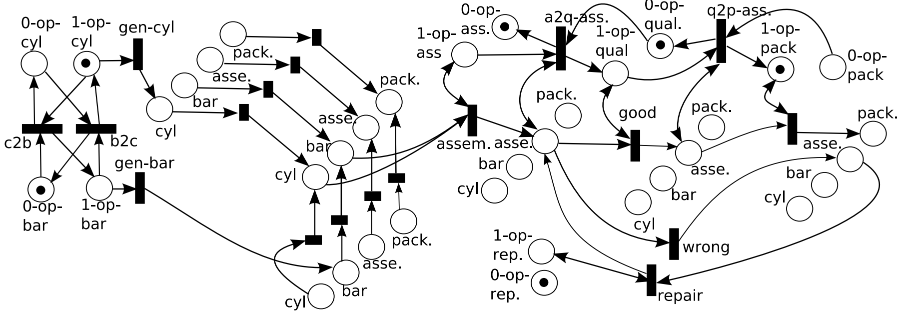

Fig. 8. Second Step in the Transformation (Some Transitions Omitted for Clarity)

4.1 Transforming the Rules Execution Control

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 rule are applicable. With this control scheme, no further transformations are needed, and in the example, the resulting Petri net is the one in Fig. 8. However, it is also possible to translate rule control structures. For example, one can assign priorities to rules [8], such that rules with higher priorities are executed first. If more than one rule has the same priority, one is executed at random. Each time a rule is executed, the control goes back to the highest priority. When no rule in a given priority can be executed, the control goes to the next lower priority. 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, and we illustrate the translation with the scheme shown in Fig. 9. The figure assumes two rules ($r1$ and $r2$) with the highest priority (priority one). These transitions, in addition, would be connected to the pre- and post- condition places, resulting from the previous step in the transformation. The idea is that in priority 1, 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 are 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 take care of removing the intermediate tokens from $r1$, $r2$, $\neg r1ex$ and $\neg r2ex$. Of course, a rule for the original DSVL can be transformed into many Petri net transitions, one for each possible match. The resulting transitions are given the same priority as the original rule.

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 possible only if the places associated with the rule are bounded. Thus, in the case of the example of previous sections, we cannot test whether

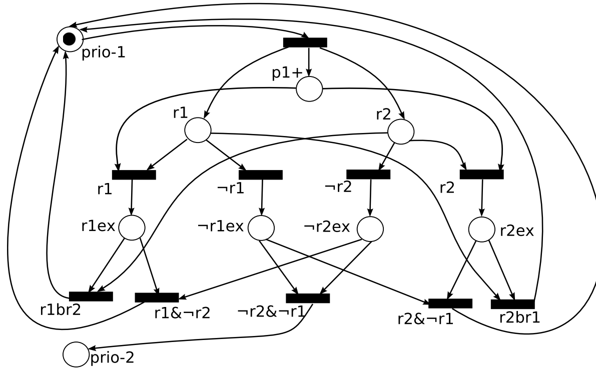

Fig. 9. Scheme for Transforming a Control Structure Based on Priorities

rules “assemble”, “move” or “change” cannot be fired, since the number of pieces in conveyors is not bounded.

Fig. 10 shows examples of the construction of the transitions for testing non-executability of a rule. Rule “rest” deletes an operator, while rule “work” models the creation of a new operator in an unattended machine. 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). Note that the “¬rest” transition makes use of the zero-testing place. TGG rule “create ¬work” creates a transition that can fire when the machine has an operator, and therefore rule “work” cannot be fired. The generated transitions can only by fired if the original rule cannot, and the firing does not produce any other effect.

Note that these kinds of TGG rules cannot be generated if the original rule has more than one NAC involving token-like elements, or a NAC and a pre-condition, both containing token-like elements, or a pre-condition with more than one token-like element. The reason is that in these cases we cannot test the non-executability of the rules in just one step, we need more than one transition. This is feasible using several transition firings, but a more sophisticated scheme than the one in Fig. 9 is needed, which we leave for future work.

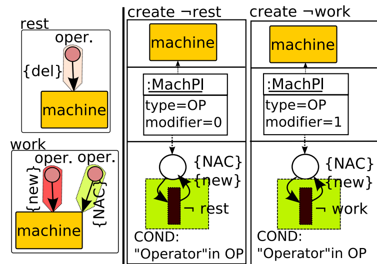

Fig. 10. Generation of Rules for Testing Non-Applicability

Typical control structures in graph transformation, such as layers, can be transformed in a similar way as priorities. For layers, the only difference is that when a rule in a layer is executed, the control remains in the current layer and does not go back to the first layer. Note that the transformation of the control structure can be kept independent of the two previous transformation steps. We are thus in effect *weaving* two transformations.

5 Algorithms for the Construction of the TGG Rules

This section gives the details for the construction of the TGG rules.

TGG Rules for the Static Information. In order to construct the TGG rules to transform the static information (like those in Fig. 4), we first explicitly copy the reference edges through the inheritance hierarchies in the meta-model triple. Thus, in the meta-model triple of Fig. 3, we add references from "MachPl" to each subclass of "Machine", from "PTok" to each subclass of "Piece", from "Place" to each subclass of "ToPlace" and from "Token" to each subclass of "ToToken". A similar closure is performed for the normal associations in the upper part of the meta-model triple (the meta-model corresponding to the DSVL).

Fig. **??** shows the approach for the generation of two of the TGG rules. We seek all possible instantiations (injective matches) of the pattern to the left in the meta-model triple (where node Z depicts a concrete class), and we generate the two rules to the right for each occurrence. The first rule adds one place for each instance of each place-like entity in the meta-model. Function *supers* returns all the superclasses of a given class. The second rule sets the initial marking of the place related to each place-like instance connected with a token-like instance. The condition checks that the name of the type of the token-like entity is included in attribute "type". For simplicity, we do not use the abstract syntax of class diagrams in the meta-model triple.

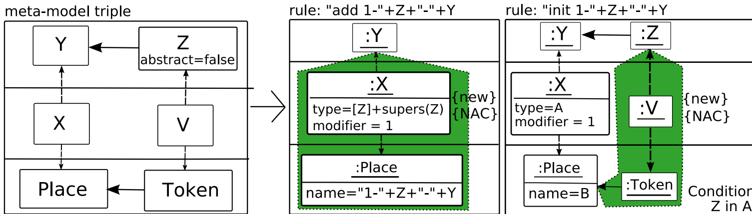

Fig. 11. Constructing the Rules for Translating the Static Information

Additional rules (similar to "add 0-op-machine" and "init 0-op-machine" in Fig. 4) are constructed for creating a zero-testing place for the bounded token-like entities. The pattern is similar to the one in the figure, but looks for a "0..1" multiplicity in the association connecting the token-like entity to the place-like entity (to the side of the latter).

TGG Rules for the Dynamic Behaviour. In addition, a TGG rule is constructed for each rule of the source DSVL. As stated before, we consider rules with an arbitrary number of NACs, but each with at most one token-like element. The construction algorithm proceeds as follows:

1. Initialize the upper part (i.e., corresponding to the source DSVL) of the TGG rule with all the place-like elements (and the connections between them) of the source DSVL rule that are tagged *NAC*, *del* or untagged. Fig. 12 shows this first step for rule "work" (shown in Fig. 10).

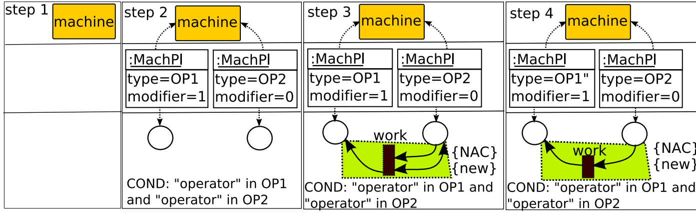

Fig. 12. Steps for Deriving the TGG rule from Rule "work"

2. For each element in the upper part of the TGG rule which was associated with a token-like element in the original rule (with tags *new*, *del* or untagged), add a mapping and a place in the middle and lower sections. Add an attribute condition stating that the type of the token-like entity is included in attribute "type" of the corresponding mapping object. If the token-like entity is bounded, or was marked as "NAC", then add an additional mapping identifying the associated zero-testing place. Do not add a mapping twice to the same place. In Fig. 12 we do not add place "1-op-machine" or the mapping twice, even when the operator appears twice in the original rule (tagged *new* and *NAC*).
3. Add a Petri net transition in the lower part of the TGG rule. Connect it to each place added due to a token-like entity marked as *new* in the original rule. Conversely, connect each place added due to a token-like element marked as *del* in the original rule to the transition. Connect the transition with a loop to each zero-testing place coming from a token-like element tagged *NAC* in the original rule. Moreover, for each connection starting or departing from the place associated with a bounded element, add the reverse connection to the associated zero-testing place. Add a loop to the transition for each place added due to an untagged token-like entity in the original rule.

   Tag the Petri net transition and the created connections in the TGG rule as *new* and *NAC*. In Fig. 12, we create a connection to place "1-op-machine" as the operator is tagged *new*. We create a loop to the zero-testing place, as the operator is marked *NAC*. Finally, we add the connection from the zero-testing place because the operator is bounded, and we added the reverse edge to the other place.
4. Simplify connections to/from the Petri net transition to zero-testing places. An incoming edge can be cancelled with an outgoing one. If a loop remains, it can be eliminated only if the place is related to a token-like element which was not marked *NAC* in the original rule (this is to allow rewriting of token-like entities by a single rule, but to retain the semantics of NACs). In the example we can cancel one outgoing and one incoming edge.
5. NACs of the original rule involving only place-like elements are copied into the TGG rule.
6. If the original rule has NACs involving both place-like and token-like elements, create an additional TGG rule following the previous steps, but

ignoring the token-like elements connected to the place-like elements in the NACs (see rule "create move-2" in Fig. 7).

TGG Rules for Testing Non-Executability. These rules generate transitions that can be fired if the original rule cannot be executed at a certain match. The procedure for their construction is similar to the previous one. The first two steps are the same. In step 3, we neglect each elements tagged *new*. Then, for each element tagged *del* or not marked, we create a self-loop from its associated zero-testing place to the Petri net transition. For each element marked *NAC*, we create a self-loop from its related place to the transition. See the rule in Fig. 10.

6 Discussion and Comparison with Related Work

Many contributions in the field of model-to-model transformation have concentrated on devising high-level means to express them. On the more formal side, we can find the seminal work on TGGs [15], which proposed an algorithm to generate operational rules (deriving for example source-to-target or target-to-source translations) from declarative ones. Recent work tries to provide even higher-level means to express the transformations, for example using *triple patterns* [10] from which operational TGG rules are generated. This is closely related to the notion of "model transformation by example" [17] (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).

However, our work is very different from these, as we express the semantics of the graph grammar rules (which express the operational semantics of the source model) with Petri nets. Petri nets can be seen as a restricted kind of graph grammar, as the token game can be considered as a graph transformation step on discrete graphs. Some work has tried to encode graph transformation rules in Petri nets, and then use the analysis techniques of the latter to investigate the former. For example, in [18] a graph transformation system is abstracted into a Petri net to study termination. However, there are several fundamental differences with our work. First, 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. Second, 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 and that we use TGGs that create mappings to the Petri net model allows us to retain the source model topology, thus the transformation does not lose information (the obtained Petri net perfectly reflects the semantics of the original language). This is thanks to the fact that a Petri net transition is constructed for each possible application of the original rule. Finally, we consider control structures for the rules and abstract rules.

In [5], graph grammars are defined for transforming DSVL models into Petri nets, without explicitly considering the original DSVL rules. Then, the

transformations are applied to the DSVL rules themselves, resulting in grammar rules simulating the Petri net. Our approach is different as we translate the DSVL rules into transitions, accurately reflecting the source DSVL semantics.

Note that we cannot translate arbitrary behavioural specifications. The source DSVL and its semantics are constrained by the following:

– The DSVL has to include elements that can be mapped to places and tokens.
– For the case of P/T nets as the target language, rules cannot create or delete place-like entities, as this would change the topology of the target model. We would need reconfigurable Petri nets [11], for example.
– Moving token-like entities (i.e., deleting and creating the edge connecting the token-like entity to the place-like entity instead of deleting and creating the edge and the entity) is possible if the target notation is place/transtion Petri nets (as we have shown when moving the operator). However care should be taken if tokens have distinct identities such as in Coloured Petri nets.
– Token-like entities are usually required to be bounded. If rules have NACs, then all token-like elements in the NAC should be bounded. Boundedness is also necessary if we are translating control structures like layers or priorities.
– NACs may have at most one token-like element. Restrictions w.r.t. the number of NACs (involving token-like elements) a rule may have, and the number of token-like elements in the pre-conditions also apply for generating negative tests. However, rules may have arbitrary NACs involving place-like elements only, as they are translated into NACs for the TGG rules and do not involve checking for tokens at run-time.

7 Conclusions

We have presented a new technique for the automatic generation of transformations into a semantic domain given a rule-based specification of the operational semantics of the source DSVL. The presented technique has the advantage that the language designer has to work mainly with the concepts of the source DSVL, and does not have to provide directly the model-to-model transformation (which can become a complex task) or have deep knowledge of the target notation.

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, TGG rules are generated that perform the transformation. Once the transformation is executed, the Petri net can be simulated or analyzed, for example to check for deadlocks or state reachability. Thus, by using 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).

We are working on tool support for this transformation generation, as well as studying other source and target languages. Moreover, we believe that for P/T nets the roles played by the source DSVL elements can be inferred by analysing the source rules (checking the static and the dynamic elements). It will also

be interesting to study how graph grammar analysis techniques are translated into P/T nets and viceversa (e.g., rule conflicts can be analysed by studying transition persistence).

Acknowledgements. Work sponsored by the Spanish Ministry of Science and Education, project MOSAIC (TSI2005-08225-C07-06). We thank the referees for their useful comments.

References

1. AGG, `http://tfs.cs.tu-berlin.de/agg/`
2. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
3. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific, Singapore (1999)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Heidelberg (2006)
5. Ermel, C., Ehrig, K.: Simulation and Analysis of Reconfigurable Systems. In: Proc. AGTIVE 2007, pp. 261–276 (to appear)
6. Guerra, E., de Lara, J.: Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages. In: SoSyM, vol. 6(3), pp. 317–347. Springer, Heidelberg (2007)
7. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
8. de Lara, J., Vangheluwe, H.: Defining Visual Notations and Their Manipulation Through Meta-Modelling and Graph Transformation. JVLC 15(3-4), 309–330 (2004)
9. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theor. Comput. Sci. 376(3), 139–163 (2007)
10. de Lara, J., Guerra, E., Bottoni, P.: Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules. In: Proc. GT-VMT 2007. Electronic Communications of the EASST, vol. 6 (2007)
11. Llorens, M., Oliver, J.: Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. IEEE Trans. Computers 53(9), 1147–1158 (2004)
12. Marriott, K., Meyer, B., Wittenburg, K.: A survey of visual language specification and recognition. In: Theory of Visual Languages, pp. 5–85. Springer, Heidelberg (1998)
13. Mellor, S., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley, Reading (2004)
14. Peterson, J.L.: Petri Net Theory and the Modelling of Systems. Prentice-Hall, Englewood Cliffs (1981)
15. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

16. Vangheluwe, H., de Lara, J.: Automatic Generation of Model-to-Model Transformations from Rule-Based Specifications of Operational Semantics. In: DSM 2007 workshop, Tech.Rep Univ. Jyväskilä (2007)
17. Varro, D.: Model Transformation by Example. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
18. Varro, D., Varro-Gyapay, S., Ehrig, H., Prange, U., Taentzer, G.: Termination Analysis of Model Transformations by Petri Nets. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 260–274. Springer, Heidelberg (2006)