Formal Aspects of Computing 22.3-4 (2010): 297–326

DOI: http://dx.doi.org/10.1007/s00165-009-0114-y

Under consideration for publication in Formal Aspects of Computing

Automating the Transformation-Based Analysis of Visual Languages

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

[1]Polytechnic School,
Universidad Autónoma (Madrid, Spain),
[2]School of Computer Science
McGill University (Montréal, Canada)

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 in the form 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. We also tackle the description of timing aspects in graph transformation rules, and its analysis through their automatic translation into Time Petri nets.

Keywords: Domain Specific Visual Languages, Graph Transformation, Model-to-Model Transformation, Time, Petri Nets, Time 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) [KeT08]. 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,

Correspondence and offprint requests to: Juan de Lara, Polytechnic School, Universidad Autónoma de Madrid, Campus Cantoblanco, C/ Francisco Tomás y Valiente, 11, 28049 Madrid (Spain) e-mail: jdelara@uam.es

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 [EEP06, LaV04], 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 [EEK99] is one such rule-based technique. One of the most commonly used formalizations of graph transformation is based on category theory [EEP06] and supports a number of interesting analysis techniques, such as detecting rule dependencies and conflicts [AGG09, EEP06, HKT02]. 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) [Mur89, Pet81]. In this case, the high-power analysis is thanks to the fact that P/T nets are less expressive than graph transformation.

For DSVLs in certain application areas, like in the real-time, embedded, or network domains, adding timing information in the language semantics becomes crucial. Timing aspects, like durations and delays, are needed in order to evaluate performance, calculate end-to-end execution delays, and throughput. However, modelling in these areas usually resorts to low-level notations, like Petri nets with time [ABC95, CeM99, Mer74, Ram74] or timed automata [BeY04, UPP09]. Even though analysis techniques and tools are available for these notations, they require expertise and their low-level nature lacks the intuitive concepts of the application domain. However, very few attempts are found trying to add timing aspects to higher-level, more intuitive notations for expressing semantics [AMP07, GHV02, Hec05, UML07, UML05].

To address the lack of analysis capabilities of high-level notations, a common technique for expressing the semantics of a DSVL is to specify a mapping from the source DSVL into a semantic domain [HKT02, LaV04] 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 analyzing the source models. However, this approach is sometimes complicated and requires from the DSVL designer deep knowledge of the semantic domain 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 [EEK99]. 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 [AGG09] or priorities [LaV04]), as well as timing information in the form of delay intervals. The main idea of this paper is to automatically generate triple graph grammar (TGG) rules [Sch94] 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 and Time Petri nets [Mer74] as the target language, but other formalisms with an explicit representation of a "simulation step" or transition (such as Constraint Multiset Grammars [MMW98] 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.

Thus, the contribution of this paper is twofold: (i) the automatic generation of model-to-model transformations from rule-based specifications of operational semantics; and (ii) the extension of graph transformation rules with time intervals, and subsequent analysis using Time Petri nets. A preliminary version of some parts of this work appeared in [LaV08].

Paper organization. Section 2 presents the rule-based approach for specification of behaviour by means of a DSVL for production systems. Section 3 gives a brief introduction to Petri nets. Section 4 shows how the initial model (i.e., the static information) is transformed. Section 5 presents the approach for translating the rules and the control structure. The algorithms used for the translation in the previous two sections are shown in the appendix, together with a correctness proof sketch. Section 6 shows how to use Petri net techniques for the analysis of the original rules. Section 7 presents our approach to add time to graph transformation rules, and how to use the previous techniques for its analysis using Time Petri nets. Section 8 presents related research and finally, Section 9 ends with the conclusions.

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. Fig. 1 shows a meta-model for the example language. It

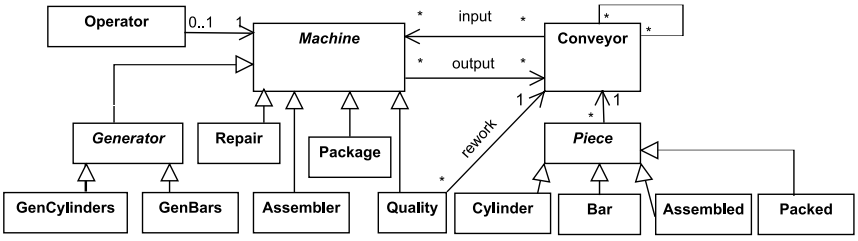

Fig. 1. Meta-Model for the Example Language.

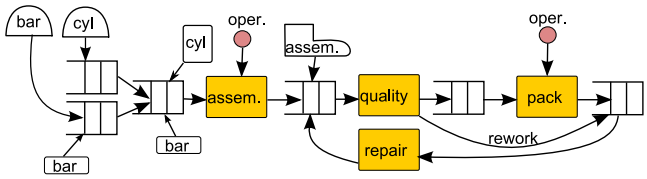

Fig. 2. Production Plant Example Model.

contains different kinds of machines (all concrete subclasses of `Machine`), 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. The latter can be inter-connected.

Fig. 2 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 an assembler and a packaging machine respectively.

Fig. 3 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 (called a *match*), then the elements marked as "{del}" are deleted, and the elements marked as "{new}" are created. This step is called a *direct derivation*. Note that even if we depict rules using this compact notation, we use the Double Pushout (DPO) formalization [EEP06] 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. As an example, Fig. 4 shows the result of applying the rule to the model in Fig. 2. In the resulting model, the derivation created an `Assembled` piece and deleted a `Cylinder` and a `Bar`.

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 rule we use *abstract objects* (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 [LBE07]. 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

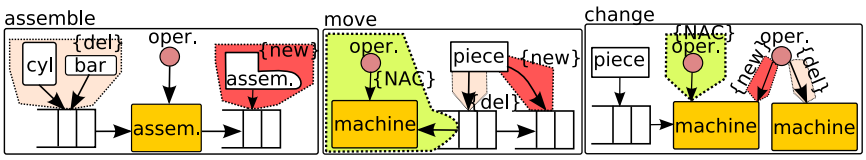

Fig. 3. Some Rules for the Production Systems DSVL.

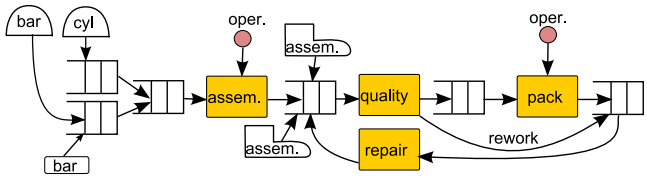

Fig. 4. Resulting Model After Applying Rule "assemble".

abstract and equivalent to 144 concrete rules. Note that this rule creates an element (the operator) together with an association which has an upper bound of 1. Here the NAC ensures that, after rule application, such bound is not violated. Similar to [TaR05], in this paper we assume that the meta-model cardinality constraints generate implicit rule post-conditions so that, should the rule application violate some of such constraints, then the rule execution is forbidden. Thus, with this assumption, the NAC in rule "change" would not be necessary, but however explicitly shows the interaction of the cardinality constraints and the rule actions (i.e., adding a bounded element induces an implicit negative post-condition). Additional rules, not shown in the paper, model the behaviour of the other types of machine.

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. There 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 5.2.

Note that the presented rules model the behaviour of machines, conveyors and operators in an untimed way. That is, if a certain rule is applied, the exact time when the direct derivation took place is not known. Only the order of application is considered at this level of abstraction. In Section 7 we present an extension of graph transformation rules to explicit handle timing information in the form of intervals.

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 it 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, in this paper we show how to automatically obtain a transformation into P/T nets starting from the previous rule-based specification if that specification is somehow "Petri net-like". P/T nets have a rich body of theoretical results and tools to analyze the previously mentioned properties. The next section presents a brief introduction to the relevant concepts of Petri nets.

3. Petri Nets

Petri nets [Mur89, Pet81] are a popular modelling language, broadly used to describe systems whose dynamics exhibit properties of distributed environments, like concurrency, synchronization, mutual exclusion, conflict and non-determinism. They incorporate a notion of distributed state, as well as a rule for state change, thus capturing both the structure and the dynamic behaviour of the real system [ABC95]. Their popularity is due to their graphical representation – nets can be visualized as a bipartite graph – as well as its solid body of theoretical results and tools enabling their analysis.

A Petri net is made up of places and transitions. The former contain tokens, and can be connected to transitions, and these to places. Intuitively, the number of tokens in places represent the system state. Transitions are the dynamic elements, and their *firing* represent a state change. Formally, a Petri net can be defined [Mur89] as a tuple $PN = (P, T, W^+, W^-, M_0)$ consisting of a finite set P of places; a finite set T of transitions; the incidence functions $W^+ \colon T \times P \to \mathbb{N}_0$ and $W^- \colon P \times T \to \mathbb{N}_0$ that represent the connection between transitions and places, and places and transitions (a zero indicates no connection, and a value greater than zero the connection weight); and an initial marking $M_0 \colon P \to \mathbb{N}_0$. The set of transitions and places should be disjoint, $P \cap T = \emptyset$ and the empty net is not allowed, $P \cup T \neq \emptyset$. We use the notation $\bullet t$ for the set of pre-places connected to t: $\bullet t = \{p \in P | W^-(p,t) > 0\}$, and $t\bullet$ for the set of post-places to which t is connected: $t\bullet = \{p \in P | W^+(t,p) > 0\}$.

As an example, the left diagram in Fig. 5 shows the graphical representation of a Petri net modelling

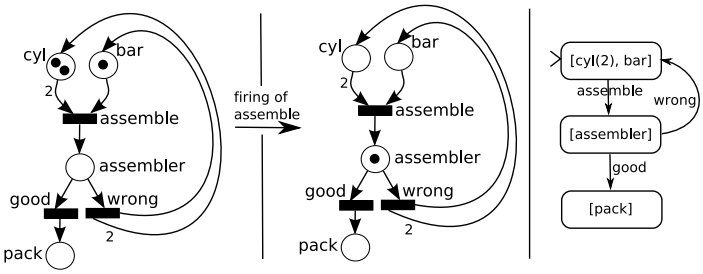

Fig. 5. A Petri Net Example (left). Firing a Transition (center). The Reachability Graph (right).

a production system. Places are represented as circles, transitions as solid rectangles and tokens as black dots inside places. The net is made of four places, $P = \{cyl, bar, assembler, pack\}$, three transitions $T = \{assemble, good, wrong\}$ and eight connections, all with a weight equal to one, except $(cyl, assemble)$ and $(wrong, cyl)$ which have a weight of two. The initial marking assigns two tokens to place *cyl*, one to *bar* and zero to the rest. Intuitively, the net represents an assembler machine that takes as input two cylinders and one bar. The net models that the machine can finish correctly and produce one *pack* element, or go wrong and return the cylinders and the bar to the input.

The state (i.e., the marking) of the net changes according to the following rule. A transition is *enabled* if and only if all its input places have at least as many tokens as the weight of their connection to the transition. Formally, iff $\forall p \in \bullet t, M(p) \geq W^-(p,t)$. In the example of Fig. 5 transition *assemble* is enabled, while *wrong* and *good* are disabled. An enabled transition *can* fire, changing the marking (from M to M') according to the following rule, $\forall p \in \bullet t \cup t\bullet : M'(p) = M(p) + W^+(t,p) - W^-(p,t)$. That is, from the input places are removed as many tokens as the weight and to the output places are added as many tokens as the weight. We use the notation $M \overset{t}{\Rightarrow} M'$ to denote that firing t changes the marking from M to M', and $M \overset{\sigma}{\Rightarrow}_* M'$ (where σ is a sequence $t_1; t_2; ...; t_n$ of transition firings) to denote the marking state changes produced by the firing sequence σ. We may omit σ and write $M \Rightarrow_* M'$, which simply means that there is some sequence able to produce M' from M (hence M' is *reachable* from M). Finally, we also use a function called *enabled* that, given a marking, returns the set of enabled transitions.

The center of Fig. 5 shows the net after firing transition *assemble*. Notice that two tokens are removed from *cyl*, one from *bar* and one is added to *assembler*, so that in general the firing of a transition does not preserve the number of tokens. Note also that, after firing *assemble* both *good* and *wrong* become enabled. Thus, one has to be chosen to be fired, and firing one disables the other, which is called a conflict.

In addition to simulation, Petri nets can be analyzed [Mur89], for example regarding reachability (whether the net can reach a certain marking), boundedness (whether the number of tokens in each place does not exceed a number k in each reachable marking), deadlocks (whether there are markings with no enabled transitions), liveness of transitions (the degree to which each transition can be fired), place invariants (whether some equations regarding the number of tokens in each place hold for any reachable marking), transition invariants (whether there is some sequence of firings that leave the marking unchanged), reversibility (whether the net can come back to the initial marking from any reachable one) and persistence (if firing transitions do not disable other enabled ones). We will review some of these properties later on in section 6.

There are several methods for analyzing these properties [Mur89]. One of them consists of generating the so called *reachability graph*, which is a representation in the form of graph of the reachable markings. The nodes in the graph represent markings, while edges correspond to firing of transitions. The right of Fig. 5 shows the reachability graph for the example net. We represent the marking as a list with the name of the places containing at least one token (if they have more than one, the number of tokens is shown in parenthesis). This way, we can observe that only three different states are possible. The net can enter in a cycle (a transition invariant) by repeatedly firing *assemble* and *wrong*. The net may deadlock, because, after firing *good*, no transition remains enabled. Finally, *good* and *wrong* are in conflict, firing either of them disables the other.

Note that, in general, the size of the reachability graph is exponential on the number of places, and moreover, it can only be constructed if the net is bounded. If this is not the case, it is possible to give an approximation of it by building the *coverability tree* [Pet81]. Other analysis techniques, like the structural and the algebraic ones [Mur89] allow certain analyses of the net without generating the reachability graph.

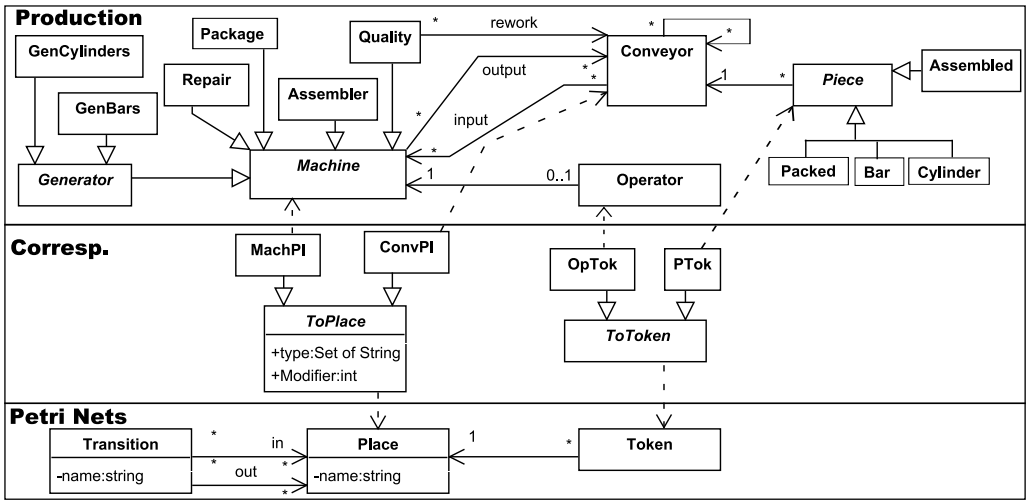

Fig. 6. Meta-Model Triple for the Transformation.

Altogether, Petri nets offer powerful analysis techniques, able to verify many interesting properties of the system. However, they lack the intuitive representation and customization degree of DSVLs. Our aim is to retain the best aspects of both approaches, so that modellers can work with the customized, domain specific notations they are proficient with, while the analysis is performed in the Petri nets domain. For this purpose, we propose a method to automatically generate a transformation from the DSVL to Petri nets, which we describe in the following sections.

4. 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, together with a dicussion sketching the proof of its correctness are left to the appendix (Section 10).

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* [GuL07], a structure declaring the allowed relations between two meta-models. A meta-model triple for the example is shown in Fig. 6. 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 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* [MSU04], 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 either of these 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 transition is reserved for the dynamic elements in the source specification: the rules modelling the operational semantics.

In order to be able to correctly express the semantics of the source DSVL with Petri nets, we need to ensure certain restrictions on the mapping (i.e., on the meta-model triple). Thus, we require each token-like entity to be connected to exactly one place-like entity, and have no more associations. This is needed to ensure that when the token-like element is deleted by some rule, the rule cannot fail due to the dangling edge condition. This restriction is shown in Fig. 7 as a pattern, similar to a graph constraint [EEP06], which

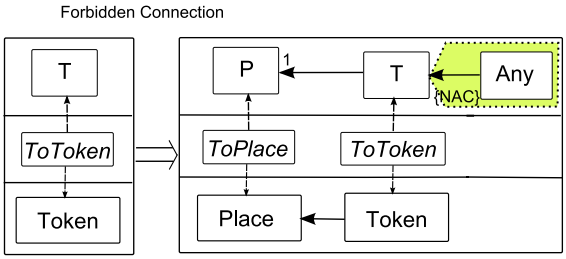

Fig. 7. Forbidden Connections from Token-like Entities in the Meta-Model Triple.

can be interpreted as an **if...then** construction. This way, if an occurrence of the left hand part is found, then such occurence has to be extended to the right hand pattern, and no occurrence of the NAC should be found. In our example, the `Piece` token-like element is connected to exactly one place-like entity (the `Conveyor` class), and the `Operator` to the `Machine` class.

Notice that we demand the relation between token-like and place-like entities to be explicitly represented with an association. In general, this could be relaxed to allow a conceptual one without explicit representation, or even using a composite relation (made of several intermediate classes and associations). Although these extensions could be possible, for simplicity, we keep the simple, explicit representation. Similarly, place-like and token-like entities could be composite elements, made of several classes and relations. Again for simplicity, we demand token-like and place-like entities to be represented by just one class. We allow the source DSVL to have elements which are neither token-like nor place-like elements, and moreover, a place-like element can receive connections from more than one token-like element.

From the meta-model triple of Fig. 6, our procedure generates a number of *operational* TGG rules [Sch94], which 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. 6).

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 the details of this construction in Section 10, here we only give some insight through examples.

Fig. 8 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 according to the DSVL meta-model). 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.

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. As will be shown later, this restricts the kind of negative tests that can be

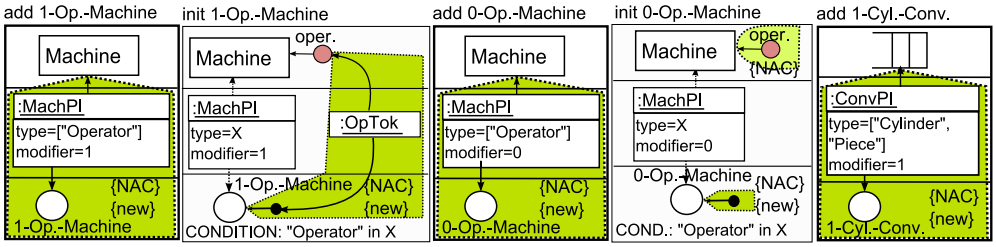

Fig. 8. Some TGG Rules for Transforming the Model.

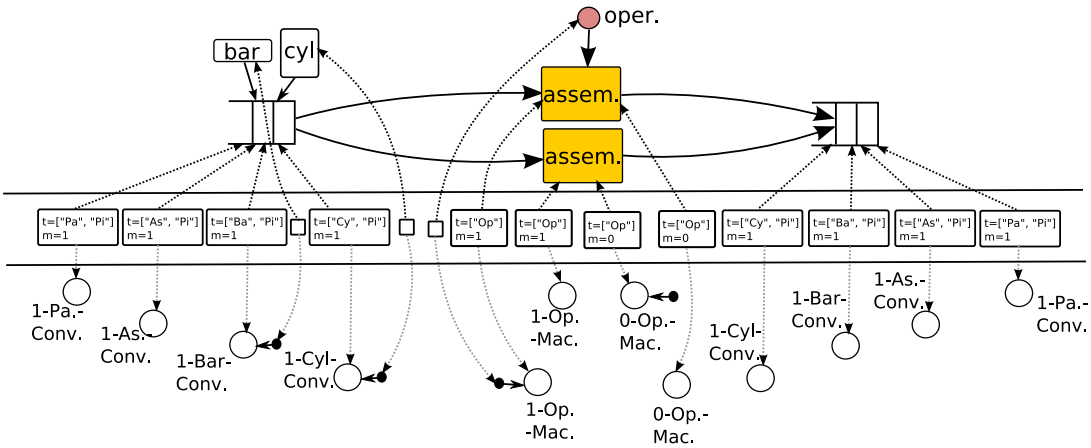

Fig. 9. First Step in the Transformation.

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 [Mur89, Pet81]. 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.

Fig. 9 shows the first step in the transformation of a simple production system made of two parallel assembler machines. The triple graph shows how each place-like entity has been linked to a place for each associated concrete token-like class. As the association between operators and machines is bounded, the latter are also linked to a zero-testing place. Finally, places are appropriately initialized with tokens, and the latter (except tokens in zero-testing places) are linked to token-like entities. Note that the correspondence graph helps not only in building the Petri net, but will also be used later to back-annotate the analysis results from the Petri net to the DSVL.

The next section shows how the translation of the dynamics is performed.

5. Transforming the Dynamic Behaviour

In order to translate the rules implementing the operational semantics (shown in Fig. 3) 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 match.

Fig. 10 shows some of the generated rules. Rule “create assemble” is generated from rule “assemble” in Fig. 3. 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

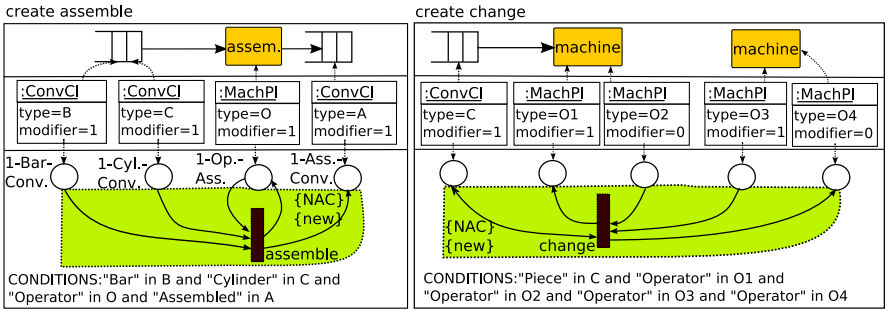

Fig. 10. Some TGG Rules for Translating the Operational Rules.

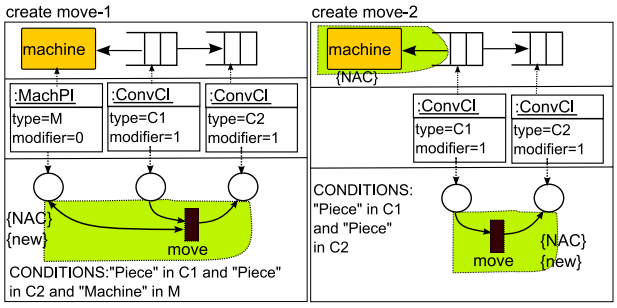

Fig. 11. Additional TGG Rules for Translating the Operational Rules.

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. 3, 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, and thus we have not created zero-testing places for them. 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 a condition in the Petri net in one step. Note that the "change" rule has indeed two negative conditions: the first is explicitly given by the NAC (forbidding the presence of operators at the target machine), the second is implied by the fact that a bounded token-like entity is added, and hence at most $k-1$ (where k is the upper bound) elements are allowed to be connected to the place-like element. In our case, the upper bound is one, and hence this implicit condition and the explicit NAC are indeed the same restriction.

Fig. 11 shows the rules generated from rule "move" in Fig. 3. 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 associated with the machine. 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 entities 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 inspecting the attribute "type" of the mapping object (this also occurred in rule *change*).

As we have seen, some restrictions apply to the rules of the source DSVL in order to be map-able onto Petri net transitions. We can summarize them in the following four restrictions:

(R1) Rules can only create or delete token-like elements. This is necessary, because in Petri nets, places are static elements. Otherwise we would need a different target formalism such as reconfigurable Petri nets [LlO04].

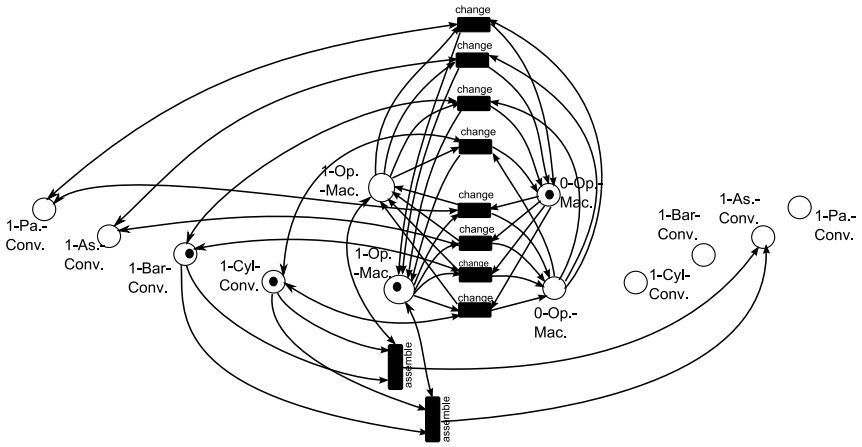

Fig. 12. Second Step in the Transformation.

(R2) Each token-like element in rules (untagged, or marked as *del*, *NAC* or *new*) should be connected to a place-like entity. In the case of deletion, this is to ensure that the rule cannot fail due to the dangling edge condition. In the other cases, this is needed to know which place-like element the token is associated with.

(R3) All token-like elements appearing in NACs should have a connection to a place-like element with bounded cardinality (that is, a "*" on the association end in the meta-model is forbidden). For example, in rules "change" and "move", operators appear in NACs, but they have bounded connections.

(R4) All NACs should have at most one type of token-like element.

Fig. 12 shows the result of applying the generated triple rules to the model in Fig. 9. For clarity, we have omitted the mappings from the Petri net places to the original model. Note that two transitions "assemble" are created as there are two different matches for triple rule "create assemble" in the triple graph of Fig. 9. They correspond to the two assemble machines. Moreover, eight "change" transitions are created, because the "create change" TGG rule seeks for any kind of piece connected to the source conveyor. As there are four types of pieces, four different matches are found to move an operator from one machine to the other one. Note that transitions created from the same rule receive the same name[1]. This is done in order to make indistinguishable transitions firings corresponding to direct derivations of the same rule but at different matches. A different label could have been chosen should we want to distinguish the match at which each rule is applied.

5.1. Optimizing the Petri Net

As a final step, the resulting Petri net can be optimized. For example, we can delete unconnected places, and places with no tokens and only a self-loop connection to a transition. Such transition can be deleted as well, as it will never be able to fire.

We have implemented such optimization rules as standard graph transformation rules (i.e., no triple graph grammar rules). The reason is that they are the same for any source DSVL, so there is no need to generate them for each different DSVL. As they are applied once the TGG transformation ends, we only consider the target Petri net model. Fig. 13 shows the rules for optimization. Rule "del place" deletes a place. As we use the DPO approach for graph transformation, the rule cannot be applied if the place has adjacent arcs or tokens. Rule "del token" removes tokens of isolated places. This way, once such tokens are deleted the place can be deleted by the previous rule. Rule "mark transition" erases a place if it is only connected to a transition with a self-loop, and the place does not have tokens. The rule creates a marking node connected to the transition that has to be deleted. The next three rules delete adjacent arcs of marked transitions, as

[1] Here we are abusing the Petri net definition given in Section 3. In fact, we are adding a transition labelling function $\lambda\colon T \to Label$ (which can be non-injective). In the following, if no confusion arises, we use $\lambda(t)$ (instead of t) to refer to transitions.

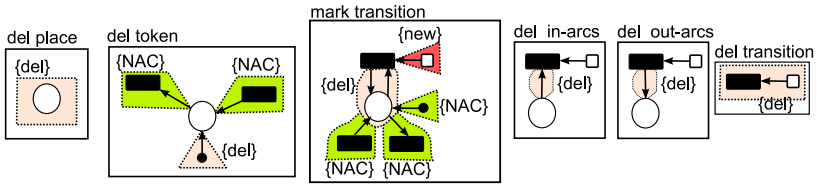

Fig. 13. Optimization Rules.

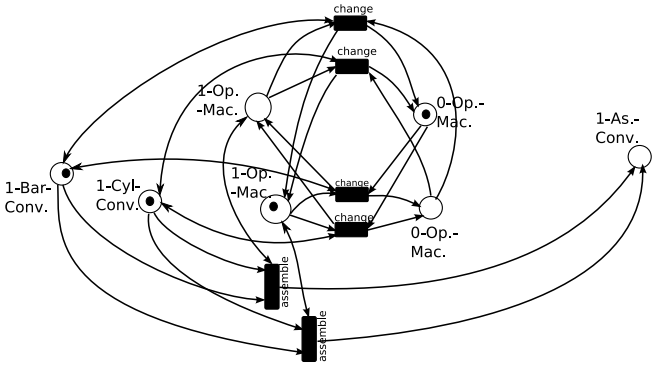

Fig. 14. Final Optimized Petri Net.

well as the transition itself. Note that the use of the marking node for the transitions to be deleted requires an expansion of the Petri net meta-model with a new auxiliary class.

Finally, Fig. 14 shows the resulting Petri net after the optimization. The rules in Fig. 13 deleted three unconnected places, as well as two places with self-loop arcs and four "change" transitions, which could never be fired.

5.2. 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 rules are applicable. With this control scheme, no further transformations are needed, and in the example, the resulting Petri net is the one in Fig. 14. However, it is also possible to translate rule control structures. For example, one can assign priorities to rules [LaV04], such that rules with higher priorities are tried first. If more than one rule has the same priority, one is executed at random. After each rule execution, the control goes back to rule with the highest priority. When no rule in a given priority class 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. 15. 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 and hence a non-deterministic choice is made. 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), represented by place *prio-2*. 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 $p1+$, $r1$, $r2$, $\neg r1ex$ 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

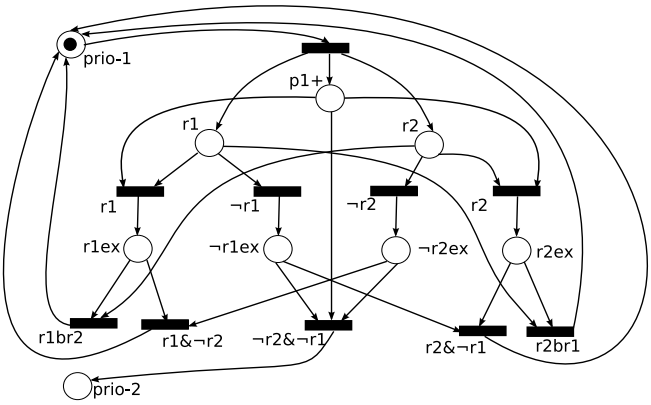

Fig. 15. Scheme for Transforming a Control Structure Based on Priorities.

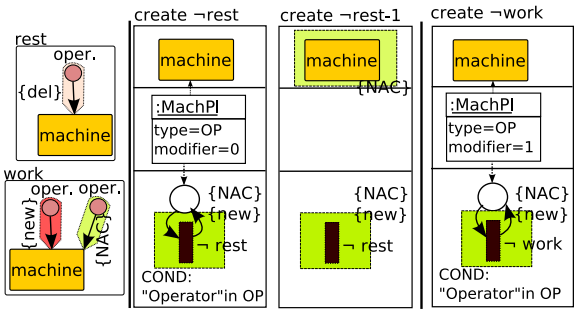

Fig. 16. Generation of Rules for Testing Non-Applicability.

with the rule are bounded. Thus, in the case of the example of previous sections, we cannot test whether rules "assemble", "move" or "change" cannot be fired, since the number of pieces in conveyors is not bounded.

Fig. 16 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. As these rules contain place-like entities and bounded token-like entities only, testing non-executability is feasible. 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. Rule "rest" may also fail to be applicable if no machine is present. As the place-like elements are static, this means that this rule would never be applicable. One may think that a TGG rule like "create ¬rest-1" is needed (creating a transition with empty input places, and hence always enabled). However this is not necessary, as if no machine exists, then no transition associated to the execution of work was created, and therefore we do not need to test if it is not applicable at that match.

Notice that rule "work" creates a bounded element (machines can have at most one operator), and that it explicitly checks that it should be absent. The TGG rule "create ¬work" creates a transition that is enabled when the machine has an operator, and therefore rule "work" cannot be fired. Notice that indeed, two restrictions are being checked here: first, the NAC demands that no operator should be connected to the machine. Second, the creation of an operator (which is bounded) requires that at least one operator can be created for that machine. That is, we cannot have `n` operators already connected to the machine, where `n` is the maximum cardinality. As in our case `n=1`, both restrictions are indeed the same and hence the loop from the place associated to the machine and the operator is able to test both conditions. Altogether, the generated transitions can only be fired if the original rule cannot, and the firing does not produce any other effect.

Note that for these negative-testing transitions to be produced, we need to add further restrictions to the DSVL rules (in addition to the four restrictions R1-R4 mentioned in section 5):

(R5) All token-like elements in the LHS of the rule must have an association to a place-like element with bounded cardinality.

(R6) The rule cannot have more than one NAC involving token-like elements.

(R7) If a token-like element appears in the LHS, then no NAC can have token-like elements.

(R8) At most one type of token-like element may appear in the LHS.

(R9) If some bounded token-like element is added, then a token-like element of the same type connected to the same place-like element may appear in some NAC, but no other token-like element can be present in the LHS or NACs.

Restriction R5 is necessary as we need the zero-testing place to check the non-existence of the token-like element. The reason for restrictions R6-R9 is that if they fail, then we cannot test the non-executability of the rule in just one step, we need more than one transition to test each possible case of failure. This is feasible using several transition firings, but a more sophisticated scheme than the one in Fig. 15 is needed, which we leave for future work.

Typical control structures in graph transformation, such as layers [AGG09, EEP06], 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, thus we are in effect *weaving* two transformations.

6. Analyzing the Petri Net

Once the system is transformed into a Petri net, it can be analyzed using standard Petri net techniques and tools [Mur89, Pet81]. We next discuss some of the properties that can be verified, using the net in Fig. 14 as an example. Most properties can be analyzed using the reachability graph (or coverability tree). Properties 2 and 3 can be analyzed using algebraic techniques, while the algebraic techniques also give necessary conditions for reachability.

To make the analysis useful for DSVL users, the verification results are not presented to the user in terms of Petri nets, but can be easily translated back to the DSVL domain [GLM09]. In general, there are three types of answers to a property analysis: (i) a yes/no answer, (ii) a sequence of transitions, and (iii) a marking. The first can be trivially back-annotated. In the second case, the transitions keep the name of the rules they were derived from. Moreover, by using distinguishing labels (see details below), it is possible to know the match (of the place-like elements) to which the rule has to be applied. Finally, for the third case, given a marking it is possible to recover a graph. First, note that the place-like elements do not change. Then, we can inspect the tokens in the net and create the corresponding token-like elements associated to place-like elements just by following the correspondence graph node mappings. The properties to be analyzed are the following:

1. **Reachability and Coverability of States**. If the net is bounded, we can investigate whether a certain state is reachable from the initial marking or not. Formally, a marking M is reachable from M_0 iff $\exists\sigma$ s.t. $M_0 \overset{\sigma}{\Rightarrow}_* M$, where σ is a sequence of transition firings.
   This property can be checked by building a reachability graph [Pet81], although a necessary condition for reachability is also obtained by using algebraic techniques. Fig. 17 shows the reachability graph for the net in Fig. 14. Using this graph, we can for example answer the question whether the system will reach a state where a piece of type `Assembled` is produced (i.e., a token in `1-As-Conv` is produced). In the example, this is the case, but not all possible execution paths lead to such a configuration, as the system may enter a cycle caused by the repeated firing of the `change` transitions. Note that we can immediately use the reachability graph to answer questions about the original graph transformation system, as a production system model can be uniquely recovered from each reachability graph state. Moreover, we have labelled with the same name, the Petri net transitions derived from a single rule, so that we abstract the particular match at which the rule was executed (but note that we could have chosen to distinguish them).
   If the net is not bounded (i.e., the number of tokens in a certain place can grow unbounded) we can approximate the reachability graph by the coverability tree [Pet81]. This structure abstracts information

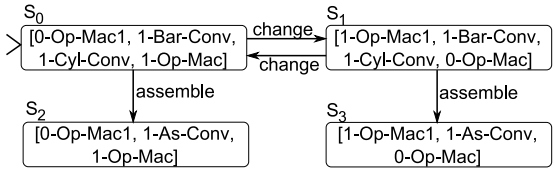

Fig. 17. Reachability Graph of the Net in Fig. 14.

by making indistinguishable two states if the possible number of tokens of one is a subset of the possible number of tokens of the other.

2. **Place invariants**. These are invariants on relations over the number of tokens of each place. They are either specified by the designer (and then checked on the reachability graph), or derived by algebraic techniques or by inspecting the reachability graph. Formally, a net with n places in initial marking M_0 is conservative with respect to the vector $(\gamma_1, ..., \gamma_n)$ iff $\sum_{i=1}^{n} \gamma_i M(p_i) = c$ for each reachable marking M (where c is a constant).
   For the example, we find that `1-Op-Mac`+`1-Op-Mac-1`= 1 (i.e., there is always one operator in the system), and that `1-Bar-Conv`+`1-Cyl -Conv`+2×`1-As-Conv = 2` (i.e., either we have one bar and one cylinder, or an assembled piece). Again, these invariants are easily translated back to the original graph transformation system – using the nodes in the correspondence graph – but note that they are relative to the initial marking (initial graph of the grammar).
3. **Transition Invariants**. These are sequences of transition firings that leave the marking unchanged (i.e., cycles). Formally, a transition invariant is a firing sequence σ such that $M \overset{\sigma}{\Rightarrow}_* M$ for some M. They can be derived by algebraic techniques or by inspecting the reachability graph.
   In the example, we find one invariant: `change; change`. The sequence moves the operator from one assembler machine to the other and back, thus reaching the same state.
4. **Reversibility**. This property, related to the previous one, checks whether the net can always reach the initial marking. If this is not the case, the system has a non-reversible process. Formally, the net is revesible if $\forall M$ s.t. $M_0 \Rightarrow_* M$, then $\exists \sigma$ s.t. $M \overset{\sigma}{\Rightarrow}_* M_0$.
   In our case, the system is non-reversible, because as soon as `assemble` is executed the net cannot reach the initial state. Again, this property is easily translated back to the original graph transformation system (this is a *yes/no* verification problem).
5. **Termination**. We can check whether for the initial marking the net always (sometimes) reaches a terminal state (a deadlock). Formally, there is a deadlock if $\exists M$ s.t. $M_0 \Rightarrow M$ and $enabled(M) = \emptyset$ (where function *enabled* returns the set of enabled transitions in a marking, see Section 3).
   For the example, the net finishes in some paths, as soon as `assemble` is fired. Note that this termination result is valid only for the given initial graph.
6. **Confluence**. This is an interesting property for graph grammars, as a grammar exhibiting termination and confluence observes a functional behaviour. Formally, a net is confluent if $\forall M, M'$ s.t. $M_0 \Rightarrow_* M$ and $enabled(M) = \emptyset$ and $M_0 \Rightarrow_* M'$ and $enabled(M') = \emptyset$ then $M = M'$. That is, if there is a unique terminal marking.
   Confluence (for the initial state) can be assessed by checking in the reachability graph whether we obtain more than two terminal states. For the example we may reach two possible terminal configurations, differing in the position of the operator, thus the grammar is not confluent.
7. **Transition persistence**. This property checks whether firing one transition disables some enabled one. Interestingly, this is related to rule conflicts and critical pair analysis in graph transformation [EEP06, HKT02, LEO06]. Two graph transformation rules are in conflict if firing one disables the other. Similarly, two Petri net transitions t and t' are in conflict if $\forall M$ s.t. $M_0 \Rightarrow_* M$ and $\{t, t'\} \subseteq enabled(M)$, then $t' \notin enabled(M')$ where $M \overset{t}{\Rightarrow} M'$ or $t \notin enabled(M'')$ where $M \overset{t'}{\Rightarrow} M''$.
   Using the reachability graph, we can make the analysis at two levels. The more abstract one uses a reachability graph like the one in Fig. 17, where we have used a non-injective labelling of transitions. In this graph we see that firing `change` does not disable `assemble` for the given initial configuration of the system, while firing `assemble` prevents `change` to be fired. A more detailed analysis can be done if we label each transition differently, even if they are generated from the same graph transformation rule. This

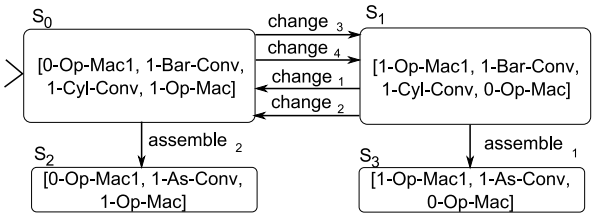

Fig. 18. Reachability Graph With Match Information.

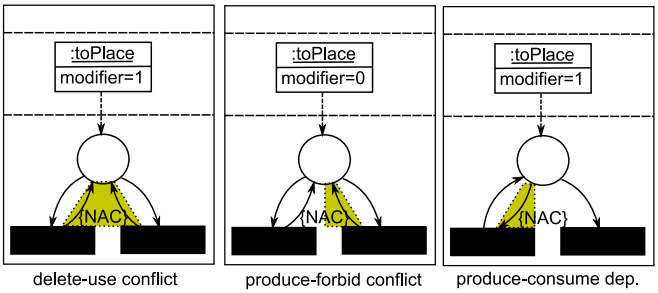

Fig. 19. Analyzing Conflicts Types and Dependencies.

way, we distinguish the match at which the rule is executed. Fig. 18 shows a detailed reachability graph, where we can see that all rules are in conflict, as not all rules in all matches remain available when one transition is fired.

The reasons for conflict can also be analyzed by inspecting: (i) places that are input to two transitions, which are not output of some of the two; (ii) zero-testing places that are input only (i.e., no output) to some transition, and which are tested with a self-loop by another one. The first kind of conflict is called *delete-use* in the graph transformation terminology, while the second one is a *produce-forbid* [LEO06]. The former arises when a rule deletes an element that another one needs, while the second is produced when a rule adds an element which is in the NAC of another one. These conflicts are represented as patterns in Fig. 19. Any occurrence of the patterns in the resulting triple graph of the transformation means a conflict. In addition, the right-most pattern is able to detect a sequential dependency between rules: one produces an element which is needed by another one.

For the example Petri net in Fig. 14, all *change* transitions have a delete-use conflict (due to places "1-Op-Mac."), as each rule moves the operator that the others need. A similar situation happens with the *assemble* transitions: they have delete-use conflicts due to places "1-Bar-Conv." and "1-Cyl-Conv.", as both transitions remove tokens that the other one needs. Transitions *change* and *assemble* are also in delete-use conflict, as *assemble* deletes tokens from "1-Cyl-Conv." and "1-Bar-Conv.", and the *change* transitions need one of the two tokens. Finally, rule *assemble* is sequentially dependent with *change* as the transitions created from the latter add a token to places "1-Op-Mac.", which are then tested by the *assemble* transitions.

Again, note that with this technique we can analyze conflicts and dependencies arising in the initial model, but not all possible conflicts. It could be possible however, like in critical pair analysis [EEP06], to generate all minimal graphs from which there is a jointly surjective match from the LHS of each two rules. Then, we can use the transformation procedure we propose to analyze conflicts with our technique.

7. Adding Time

The rules describing the operational semantics of the DSVL are untimed. That is, the execution of a certain rule sequence is ordered, but no information is given on the exact time at which the firing of each rule is performed. For some DSVLs (e.g., in the real-time domain), such information is essential. Therefore, for more realistic modelling of the behaviour of the systems, timing information is needed.

Several approaches have been proposed to add time to Petri nets. Time can be added to places, tokens,

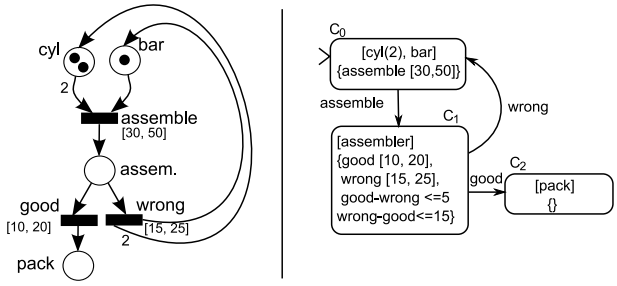

Fig. 20. A TPN Example (left). State Classes (right).

arcs and transitions [CeM99]. In the latter case, the firing of a transition is given a duration, which is a real number in Ramchandani's Timed Petri nets [Ram74], an interval in Merlin's Time Petri nets [Mer74], and a probability distribution in Stochastic Petri nets [ABC95]. Taking this analogy with Petri nets, in the original DSVL, we can add time in place-like entities, token-like entities or in the rules. The two former cases imply modelling timing as data inside the model [GHV02], while in the latter case the time is added to the formalism itself [Hec05]. Similar to Time Petri nets, here we take the approach of adding a time interval to graph transformation rules, however we could also use any of the other approaches. The advantage of using intervals is that they are a natural means to model imprecise clocks, time outs and acceptable variability of timing constraints.

We start by first briefly reviewing Time Petri Nets, then we describe how to add time to graph transformation rules, and finally we show how to use the previously presented translation procedure to map the timed rule specification into a Time Petri net for analysis.

7.1. Time Petri Nets

Time Petri nets (TPNs) [Bed91, CeM99, Mer74, WaX00] associate an interval $[a, b]$ (with $0 \leq a \leq b$, and b possibly unbounded) with each transition $t \in T$ (where T is the set of transitions). The lower limit is called the *Static Earliest Firing Time* (Static EFT), while the upper one is the *Static Latest Firing Time* (Static LFT). This interval is modelled by a function $SIM\colon T \to \mathbb{Q}^* \times (\mathbb{Q}^* \cup \infty)$ [Bed91], where $\mathbb{Q}^*$ is the set of positive rational numbers[2]. Function SIM is constrained such that for each transition t_i, $SIM(t_i) = (\alpha_i^S, \beta_i^S)$, with α_i^S and β_i^S rational values such that $0 \leq \alpha_i^S \leq \infty$, $0 \leq \beta_i^S \leq \infty$, $\alpha_i^S \leq \beta_i^S$ if $\beta_i^S \neq \infty$ or $\alpha_i^S < \beta_i^S$ if $\beta_i^S = \infty$.

Thus, the simulation of a TPN takes into consideration the time. This way, assuming that a transition t_i becomes enabled at an absolute time τ_{Abs}, it cannot fire, while being continuously enabled, before $\tau_{Abs} + \alpha_i^S$, and must fire before or at the latest time $\tau_{Abs} + \beta_i^S$. Note that in this model, firing a transition takes no time to complete. Moreover, untimed Petri nets can be seen as a special case of TPNs, where each transition t_j is assigned an interval $(\alpha_i = 0, \beta = \infty)$. Thus, TPN models are timed restrictions of the untimed net.

The left of Fig. 20 shows a TPN example modelling a simple production plant (the timed version of the one in Fig. 5). Transition `assemble`, once enabled, cannot fire before 30 time units, and should fire before or at 50 time units. Thus, this interval models the processing delay of an `assemble` machine. Transitions `good` and `wrong` are in conflict, firing the former yields to a deadlock, while the latter puts the tokens in their original places. A possible *firing schedule* is for example $(assemble, 35)(good, 50)$, but note that there are infinite firing sequences of length two leading to a state with one token in `pack`.

A TPN state $S = (M, I)$ is made of a marking M (i.e., the number of tokens in each place) and a partial function I that gives the *dynamic* EFT and LFT for each enabled transition. We use the notation (α_i, β_i) for the dynamic interval $I(t_i)$ of transition t_i. Note that, except for the initial state, the dynamic EFT and LFT of each transition will in general not be equal to its static intervals. This is because when a transition is fired, the dynamic intervals of all enabled transitions which are not in conflict are decreased. Thus, at time τ in a given state $S = (M, I)$, a transition t_i can fire at time $\tau + \theta_i$ iff:

[2] We take the approach of [Bed91], where rationals are used instead of reals. In practice, this does not impose any limitation, and theoretically is needed in order to ensure boundedness of the state classes of the TPN when the untimed net is bounded

- it is enabled at time τ (as in the untimed case), and
- if $\alpha_i \leq \theta_i \leq min\{LFT \text{ of } t_k\}$, where k ranges over the set of enabled transitions in M.

The latter condition indicates that the relative firing time of t_i, given by θ_i, must lie between the dynamic EFT of the transition and the minimum dynamic LFT of each enabled transition (otherwise it would prevent some other transition to fire).

In the example of Fig. 20, after firing transition `assemble` at time τ, both transitions `good` and `wrong` become enabled. Any of the two transitions must fire before or at $\tau + 20$, which is the LFT of transition `good`. Transition `good` cannot fire before $\tau + 10$, while `wrong` cannot fire before $\tau + 15$.

If, in state $S = (M, I)$ transition t_i is fired at time $\tau + \theta_i$, the net state changes to $S' = (M', I')$. The new marking M' is calculated as in untimed Petri nets: $\forall p \in \bullet t_i \cup t_i \bullet : M'(p) = M(p) + W^+(t_i, p) - W^-(p, t_i)$. The dynamic interval I' for each transition $t_j \in T$ is calculated as follows:

- If t_j is not enabled in M', then t_j is not present in the domain of I'.
- If t_j is was enabled in M and is still enabled in M', then $I'(t_j) = (max(0, \alpha_j - \theta_i), \beta_j - \theta_i)$. That is, the dynamic interval of t_j is shifted θ_i to the origin. Note that the upper limit of the interval is greater than or equal to zero as $\theta_i \leq \beta_j$.
- If t_j is enabled in M' but was not enabled in M, then $I'(t_j) = (\alpha_j^S, \beta_j^S)$. That is, the dynamic interval is initialized with the static interval.

In a different view, there are two aspects that modify the net state: the passing of time (which decreases the dynamic intervals of enabled transitions) and the firing of transitions.

In the example of Fig. 20, the initial state is given by $S_0 = (M_0 = \{(cyl, 2), (bar, 1), (assem, 0), (pack, 0)\}, I_0 = \{(assemble, (30, 50))\})$. This way, transition `assemble` is allowed to fire at any time in the interval $0 + [30, 50]$ and its actual firing time is allowed a potentially infinite number of values in the interval. Firing `assemble` leads to state $S_1 = (M_1 = \{(cyl, 0), (bar, 0), (assem, 1), (pack, 0)\}, I_1 = \{(good, (10, 20)), (wrong, (15, 25))\})$, where the dynamic interval function I_1 does not contain `assemble` in its domain as it is not enabled, but transitions `good` and `wrong`.

The set of reachable states of a TPN is a subset of the untimed net. Interestingly, adding intervals to transitions may prohibit certain execution paths and transition conflicts that were possible in the untimed net.

There are several analysis techniques for TPNs. For example, in [Bed91], so called *state classes* are derived. A state class is defined as the union of all firing values that are possible for a given marking, and is defined by a pair $C = (M, D)$, a marking M and a domain D. The latter is the set of solutions to the system of inequalities capturing the global timed behaviour of the TPN [Bed91]. The number of state classes is bounded iff the underlying untimed Petri net is bounded. The classes for the example TPN are shown to the right of Fig. 20. The class of the initial marking is given by $C_0 = (M_0, \{30 \leq \theta_{assemble} \leq 50\})$, which describes all valid firing times for `assemble`. When `assemble` is fired, the net goes to class $C_1 = (M_1, \{10 \leq \theta_{good} \leq 20; 15 \leq \theta_{wrong} \leq 25; \theta_{good} - \theta_{wrong} \leq 5; \theta_{wrong} - \theta_{good} \leq 15\})$. The first two inequalities account for the static intervals of transitions `good` and `wrong`, the third and fourth express the constraints of firing `wrong` and `good` first respectively. If `wrong` is fired first, we have that $\theta_{wrong} - \theta_{good} \leq 0$ and we reach class C_0, while if `good` is fired first, we reach class $C_2 = (M_0 = \{(cyl, 0), (bar, 0), (assem, 0), (pack, 1)\}, D_2 = \emptyset)$ and have $\theta_{good} - \theta_{wrong} \leq 0$.

The *clock-stamp* method of [WaX00] allows calculating the end-to-end time delay for task execution, which is useful for the verification of timing constraints in real-time systems. The technique consists on adding global time stamps to each reachable state, so as to account for the time interval in which such state can be reached. In the example of Fig. 20, the net can reach state C_1 at $[30, 50]$, C_2 at $[40, 70]$, and come back to C_0 at $[75, 125]$.

7.2. Timed Rule-Based Specification

As previously stated, we do not want to resort to low-level notations like TPNs for modelling the timed behaviour of a DSVL, but we want to provide the designer a means to incorporate the timing information already in the rule-based specification. Then, using the translation we have presented in previous sections, the rule-based specification will be transformed into a TPN for analysis.

Using an analogy with TPNs, we provide each rule r_i with a static time interval (α_i^S, β_i^S). This means that

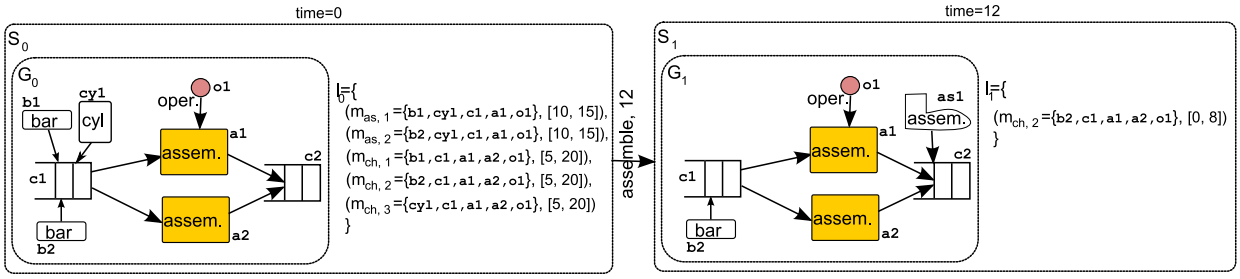

Fig. 21. One Step in a Timed Rule-Based Simulation.

a rule r_1 for which a match m_1 is found at time τ_{Abs}, and for which such match is continuously maintained, cannot be executed before time $\tau_{Abs} + \alpha_i^S$ and should be executed no later than or at time $\tau_{Abs} + \beta_i^S$. Thus, a time graph grammar is defined as a tuple $GT = (G_0, R = \{r_i\}_{i \in I}, SIM\colon R \to \mathbb{Q}^* \times (\mathbb{Q}^* \cup \infty))$, where G_0 is the initial graph, R is the set of rules, and SIM is the function assigning to each rule r_i its static time interval (α_i^S, β_i^S).

In our DSVL example, the time intervals allow modelling timing constraints of the actions represented by each rule. For example, assigning interval $[10, 15]$ to rule `assemble` in Fig. 3 means that, when an occurrence is found for such rule, a delay of at least 10 and no more than 15 should elapse before it can be executed. Thus, the interval models the execution time of machine `assemble`. Similarly, assigning interval $[2, 4]$ to rule `move` accounts for the time it takes a piece to move between conveyors, and interval $[5, 20]$ models the time interval for an operator to move between machines.

Note that the semantics of graph transformation with time is similar to TPNs, but the dynamic intervals are assigned to matches (i.e., occurrences of the rule's LHS in the host graph) and not to rules themselves. This way a graph transformation state $S_j = (G_j, I_j = \{(m_{ik}\colon L_i \to G_j, (\alpha_{ik}, \beta_{ik}))\})$ is made of a graph G_j and a set of pairs where the first element $m_{ik}\colon L_i \to G_j$ is a valid match of rule r_i in graph G_j (L_i is the LHS of rule r_i), and the second the dynamic interval assigned to match m_{ik}. The set I_j contains one element for each valid match from each rule $r_i \in R$ in G_j. As an example, Fig. 21 shows to the left the initial state S_0 of a simulation. There are two matches of rule `assemble` ($m_{as,1}$ and $m_{as,2}$, indicated by node identifiers), and three of rule `change` (as the abstract node `piece` in the rule can get matched to any of the three pieces in the model). The matches are annotated with their dynamic interval. As this is the initial state, the dynamic interval of each match is equal to the static one.

In state S_j at time τ_j, a rule r_i can be executed at a certain match m_{ik} at time $\tau_j + \theta_{ik}$ iff $\alpha_{ik} \leq \theta_{ik} \leq min\{LFT \text{ of } m_r\}$, where α_{ik} is the lower bound of the dynamic interval of match m_{ik}, and m_r ranges on all current matches in the set I_j. When the rule is applied at time $\tau_j + \theta_{ik}$, the graph G_j is modified according to the normal (untimed) direct derivation semantics, yielding G_{j+1}, and the dynamic interval I_{j+1} is obtained as follows:

- If a match m_r is not present anymore in G_{j+1}, then it is removed from I_{j+1}.
- If a match m_r is preserved by the direct derivation, then $I_{j+1}(m_r) = (max(0, \alpha_r - \theta_{ik}), \beta_r - \theta_{ik})$.
- For every new match m_{hq} of any rule $r_h \in R$ in the grammar, $I_{j+1}(m_{hq}) = (\alpha_h^S, \beta_h^S)$, where (α_h^S, β_h^S) is the static interval of rule r_h.

As an example, Fig. 21 shows the firing of rule `assemble` at time 12 starting from state S_0. Only the match $m_{ch,2}$ of rule `change` is preserved, so its dynamic interval is decreased in 12 units. In the resulting state S_1, the rule `change` must be fired at match $m_{ch,2}$ before or at 8 time units.

Similar to TPNs, the set of reachable states of a time graph grammar is a subset of the reachable states of the untimed grammar.

7.3. Translation into TPNs and Analysis

We can use the procedure described in sections 4 and 5 for the translation from a timed rule specification into TPNs. The only difference is that, in the timed case, the intervals assigned to the rules are copied to the Petri net transitions by the TGG rules described in Section 10.2. Additional optimization rules can be

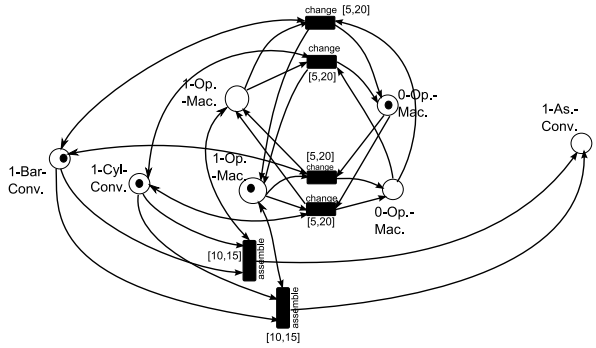

Fig. 22. Resulting TPN.

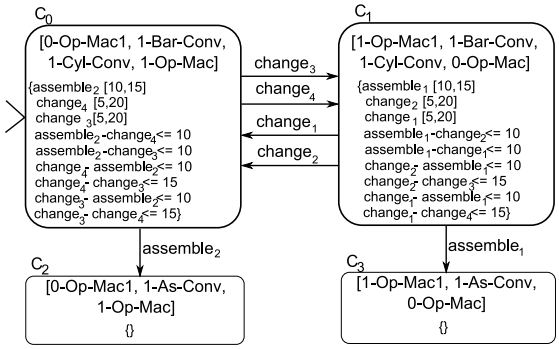

Fig. 23. State Classes of the TPN of Fig. 22.

derived, though. For example, if two transitions t_i and t_j are in conflict, and the static EFT of t_i is bigger than the static LFT of t_j, then we can remove t_i (and all its adjacent arcs) as t_j will always be fired first, disabling t_j, which will never be able to fire.

Fig. 22 shows the resulting TPN obtained from the production system of Fig. 9. The structure of the net is the same as the resulting net from the untimed graph transformation rules, but time intervals have been added to the transitions according to the rule intervals.

Fig. 23 shows the state classes corresponding to the generated TPN. It can be observed that the number of classes corresponds to the number of states of the untimed reachability graph, as no transition remains enabled after firing any transition, and the time intervals do not prevent any transition to fire. For clarity, we have differentiated the transition labels, like in Fig. 18, but similar to the untimed case, a more abstract graph can be obtained which does not distinguish the matches. From the firing domains of the classes, we can observe that in classes C_0 and C_1 firing `change` has to occur before or at 15 time units, as otherwise `assemble` has to be fired. That is, a change will never happen at interval $(15, 20]$. Note that this analysis is automatically supported by the Romeo tool [GLM05], which also performs on-the-fly model-checking of reachability properties, as well as translation of the net into a timed automata [BeY04, UPP09].

We can also use the clock-stamp method of [WaX00] to calculate end-to-end delays for task executions. For example, the interval for two operator changes is $[10, 30]$, and for producing a piece in the currently unattended machine $[15, 30]$.

8. 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 [Sch94], 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* [LaG08] from which operational TGG rules are generated.

This is closely related to the notion of "model transformation by example" [Var06], where transformation rules are derived starting from a mapping between two meta-models, and transformation models [BBG06], 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 [VVE06] 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, as well as abstract rules.

In [KKR08], model-to-model transformations are expressed using a mapping language, whose semantics rely on Coloured Petri nets. The mapping is done at the meta-model level using operators, whose low-level implementation is a coloured Petri net. This means, that if no suitable operator is available for the transformation purpose, one has to build a new one by directly using Coloured Petri nets, a low-level task which requires specialized knowledge and expertise. On the contrary, our specification language, graph grammars, is higher-level, and we provide an automatic means for its translation into a P/T net for analysis.

In [ErE07], 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. The same authors in [EhE08] provide the first steps towards a method to verify whether behaviour is preserved by model-to-model transformations into semantic domains. A transformation is semantically correct if for each simulation run of the source DSVL there is a corresponding run in the target system. Conversely, semantical completeness is obtained if for each simulation run on the target system, there is an equivalent one in the source model. This verification approach is interesting, as it can be done at the rule level, as rules simulating the target language are derived. In our case, the simulation of the resulting net is not performed by rules, but using the normal semantics of Petri nets. In the appendix, we show that if a rule of the original DSVL is applicable at some match, then the transition generated for that match is enabled and vice versa.

With respect to adding time to graph transformation rules, only a few attempts are found. In [GHV02], time is represented as a distinguished attribute `chronos` in the graph nodes. This is similar to the Petri net approaches where time is added to the tokens, in particular to time ER nets [GMM91]. In [Hec05], an exponentially distributed application delay is associated with each rule. This approach is inspired by stochastic Petri nets [ABC95]. Interestingly, the set of reachable graphs of a certain stochastic graph grammar is the same as in the untimed grammar, as the adding of time only amounts to labeling the resulting transition system with probabilities. In our case, the adding of time has a stronger semantics, as it may suppress certain execution paths. Finally, in [SyV08] time is added to the rule execution control language, based on a discrete-event simulation formalism.

Altogether our work is original, as we are able to automatically generate a model-to-model transformation starting from rules expressing the operational semantics of the source DSVL. Moreover, we take into consideration possible control execution structures for the source grammar. Note however 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 (restriction R1 in Section 5), as this would change the topology of the target model. We would need reconfigurable Petri nets [LlO04], 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 P/T 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 (restriction R3 in Section 5). Boundedness is also necessary if we are translating control structures like layers or priorities (see restrictions R5-R9 in Section 5.2).
- NACs may have at most one token-like element (restriction R4). Restrictions with respect to 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.

With respect to portability, we believe that this approach can be used with different target languages (in addition to different kinds of Petri nets) having an explicit notion of transition. The steps to follow are the following: (i) produce a meta-model of the target language, (ii) define mappings to the relevant entities in the target language (similar to the `ToPlaces` and `ToToken` mappings in Fig. 1), (iii) study how to map the static elements of the source DSVL models into static information of the target language, and define TGG rules for this purpose, and (iv) study how to map DSVL rules into the transition elements of the target language, and define TGG rules for this task. We are currently working on identifying further appropriate target languages (possibly some types of process algebras, or multiset rewriting formalisms) and using this method to automate the translation.

Concerning the time semantics, we have made two decisions. First, when a transition gets disabled and then enabled again, its dynamic interval is initialized with the static one (this semantics is called *enabling memory* [ABC95]). That is, we are not taking into account the "work done" when the transition was enabled the previous time. Taking into account this time is called *age memory* [ABC95]. While both options are possible in principle for TPNs, age memory is difficult to emulate in time graph transformation for the general case. This is so because, when a match is disabled, it may be because of deleted elements, and thus it is problematic to decide if a new match can be considered the same match as a previously existing one. Note that we could take this decision at the rule level: if a rule gets disabled and then enabled again, we take into account the time it remained enabled the last time. However, this would not be a sensible decision, as it neglects the place in the graph where the rule gets enabled. Thus, matches in graph transformation are much more dynamic than transitions in Petri nets. However, for restricted cases of grammars, like the ones we can map to Petri nets, age memory semantics is possible, as matches can be related to place-like elements, which are not created or deleted. This way, place-like elements are the means to decide whether a match is the same as a previously existent one.

Second, the pre-places of a transition may have enough tokens to enable the transition more than once. This is related to the *server semantics* [ABC95, BoD01]. In the *single server semantics*, as soon as a transition is enabled, its timer starts to count. When the transition fires, if it is still enabled, a new timer is set. Thus, in this way, there is only one timer per transition, and tokens are processed serially. In the *infinite server semantics*, a timer is set for each set of tokens that enable a transition. There is no bound on the number of timers each transition may have, and the tokens are processed in parallel. It is possible to restrict the parallelism up to a maximum, arbitrary degree of k, which is called the *k-multiple server semantics*. In TPNs, any of these possibilities can be chosen [BoD01]. For time graph transformation the natural choice seems to be the infinite server semantics: as soon as a new match for a rule becomes available, a new timer is set. However, a single server semantics is also possible, and accounts to scheduling just one match per rule, and process each match sequentially.

9. 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 also shown how to handle time in rule-based

specifications of DSVL semantics, and we have been able to analyze such specifications by using a mapping into TPNs.

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 to perform the transformation. Once the transformation is executed, the Petri net can be simulated or analyzed, for example to check for deadlock 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 full 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 analyzing the source rules (checking the static and the dynamic elements). We are also working in handling more complex kinds of NACs, trying to relax the restrictions R1-R9 for the source DSVL, and porting this approach to other target languages to remove the need for some of these restrictions. With respect to the timing extensions to graph grammars, we are working on tool support for different semantics, and on developing additional useful concepts for performance evaluation. For example, it is possible to identify resource elements at the meta-model level, and then calculate resource utilization metrics taking into account the time intervals assigned to the rules.

Acknowledgements. Work sponsored by the Spanish Ministry of Science and Innovation, project METEORIC (TIN2008-02081/TIN) and by the Canadian Natural Sciences and Engineering Research Council (NSERC). We thank the referees for their useful comments that helped us to improve the paper.

References

[AGG09] AGG home page at: `http://tfs.cs.tu-berlin.de/agg/`.

[ABC95] Ajmone Marsan, A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G. 1995, *Modelling with Generalised Stochastic Petri Nets.* Wiley Series in Parallel Computing, John Wiley and Sons.

[AMP07] André, C., Mallet, F., Peraldi-Frati, M.-A. 2007. *Multiform Time in UML for Real-time Embedded Applications.* Proc. 13th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications, pp.:232-240.

[BeY04] Bengtsson, J., Yi, W. 2004. *Timed Automata: Semantics, Algorithms and Tools.* Proc. ACPN 2003, LNCS 3098, pp.: 87124, Springer.

[Bed91] Berthomieu, B., Diaz, M. 1991. *Modeling and Verification of Time Dependent Systems Using Time Petri Nets.* IEEE Transactions on Software Engineering, Vol 17(3), pp.: 259-273.

[BBG06] Bézivin, 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.

[BoD01] Boyer, M., Diaz, M. 2001. *Multiple Enabledness of Transitions in Petri Nets with Time.* Proc. 9th IEEE Int. Workshop on Petri Nets and Performance Models (PNPM'01), pp.: 219-228.

[CeM99] Cerone, A., Maggiolo-Schettini, A. 1999. *Time-Based Expressivity of Time Petri Nets for System Specification.* Theoretical Computer Science 216, pp.: 1-53. Elsevier.

[EEK99] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1.* World Scientific.

[EEP06] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation.* Springer.

[EhE08] Ehrig, H., Ermel, C. 2008. *Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation.* Proc. ICGT'08, LNCS 5214, pp.: 194-210. Springer.

[ErE07] Ermel, C., Ehrig, K. 2007. *Simulation and Analysis of Reconfigurable Systems.* Proc. AGTIVE'07, pp.: 261-276.

[GLM05] Gardey, G., Lime, D., Magnin, M., Roux, O. H., 2005. *Romeo: A Tool for Analyzing Time Petri Nets.* Proc. Computer Aided Verification, LNCS 3576, pp.: 418-423.

[GMM91] Ghezzi, C., Mandrioli, D., Morasca, S., Pezzé, M. 1991. *A unified high-level petri net formalism for time-critical systems.* IEEE Transactions on Software Engineering 17(2):160172.

[GuL07] Guerra, E., de Lara, J. 2007. *Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages.* SoSyM (Springer), Vol 6(3), pp.: 317-347.

[GLM09] Guerra, E., de Lara, J., Malizia, A., Díaz, P. 2009. *Supporting User-Oriented Analysis for Multi-View Domain-Specific Visual Languages.* Information and Software Technology 51(4), pp.: 769-784. Elsevier.

[GHV02] Gyapay, S., Heckel, R., Varró, D. 2002. *Graph Transformation with Time: Causality and Logical Clocks.* Proc. ICGT'02, LNCS 2505, pp.: 120-134, Springer.

[HKT02] Heckel, R., Küster, J. M., Taentzer, G. 2002. *Confluence of Typed Attributed Graph Transformation Systems.* Proc. ICGT'02, LNCS 2505, pp.: 161-176, Springer.

[Hec05] Heckel, R. 2005. *Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks.* Proc. ICTAC'05, LNCS 3722, pp.: 53-69.

[HoV87] Holliday, M. A., Vernon, M. K. *A Generalized Times Petri Net Model for Performance Analysis.* IEEE Trans. Software Eng., Vol 13 (12), 1987, 1297-1310.

[KKR08] Kappel, G., Kargl, H., Reiter, T., Retschitzegger, W., Schwinger, W., Strommer, M., Wimmer, M. *A Framework*

for Building Mapping Operators Resolving Structural Heterogeneities. Proc. UNISCON 2008, Lecture Notes in Business Information Processing 5, pp.: 158-174. Springer.

[KeT08] Kelly, S., Tolvanen, J. P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Press.

[LEO06] Lambers, L., Ehrig, H., Orejas, F. 2006. *Conflict Detection for Graph Transformation with Negative Application Conditions.* Proc. ICGT'06, LNCS 4178, pp.: 61-76.

[LaV04] de Lara, J., Vangheluwe, H. 2004. *Defining Visual Notations and Their Manipulation Through Meta-Modelling and Graph Transformation.* Journal of Visual Languages and Computing, Vol 15(3-4), pp.: 309-330. Elsevier.

[LBE07] 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.

[LaG08] de Lara, J., Guerra, E., 2008. *Pattern-Based Model-to-Model Transformation.* Proc. ICGT' 2008, LNCS 5214, pp.: 426-441.

[LaV08] de Lara, J., Vangheluwe, H. 2008. *Translating Model Simulators to Analysis Models.* Proc. FASE 2008, LNCS 4961, pp.: 77-92, Springer.

[LlO04] Llorens, M., Oliver, J. 2004. *Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets.* IEEE Trans. Computers 53(9), pp.: 1147 - 1158.

[MMW98] Marriott, K., Meyer, B., Wittenburg, K. 1998. *A survey of visual language specification and recognition.* Theory of Visual Languages. Pages 5-85. Springer-Verlag.

[MSU04] Mellor, S., Scott, K., Uhl, A., Weise, D. 2004. *MDA Distilled: Principles of Model-Driven Architecture.* Addison Wesley.

[Mer74] Merlin, P. 1974. *A Study of the Recoverability of Computer Systems.* Ph.D. Thesis, University of California, Irvine.

[Mur89] Murata, T. 1989. *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEE, Vol. 77(4). Pp.: 541-580.

[Pet81] Peterson, J. L. 1981. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall.

[Ram74] Ramchandani, C. 1974. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* Project MAC, TR 120, Massachussets Institute of Technology.

[Sch94] Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars.* Proc. WG'94. LNCS 903, pp.: 151 - 163. Springer.

[SyV08] Syriani, E., Vangheluwe, H. 2008. *Programmed Graph Rewriting with Time for Simulation-Based Design.* Proc. ICMT 2008. LNCS 5063, pp.: 91-106. Springer.

[TaR05] Taentzer, G., Rensink, A. 2005. *Ensuring Structural Constraints in Graph-Based Models with Type Inheritance.* Proc. FASE 2005, LNCS 3442, pp.: 64-79.

[UML07] UML 2.1.2 infrastructure and superstructure specification (2007): `http://www.omg.org/spec/UML/2.1.2/`

[UML05] UML Profile for Schedulability, Performance, and Time Specification v 1.1(2005): `http://www.omg.org/technology/documents/formal/schedulability.htm`

[UPP09] Home page of UPPAAL: `http://www.uppaal.com`

[Var06] Varro, D. 2006. *Model Transformation by Example.* Proc. MoDELS'06, LNCS 4199, pp.: 410-424, Springer.

[VVE06] 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.

[WaX00] Wang, J., Xu, G. 2000. *Reachability Analysis of Real-Time Systems Using Time Petri Nets.* IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics, Vol 30(5). pp.: 725-736.

10. Appendix: Algorithms for the Construction of the TGG Rules

This appendix provides the details for the construction of the TGG rules, and an informal proof sketch that shows that firing a Petri net transition is only possible if the DSVL rule it was derived from is applicable and the other way round.

10.1. TGG Rules for the Static Information

In order to construct the TGG rules to transform the static information (like those in Fig. 8), we first explicitly copy the reference edges through the inheritance hierarchies in the meta-model triple. Thus, in the meta-model triple of Fig. 6, 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. 24 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

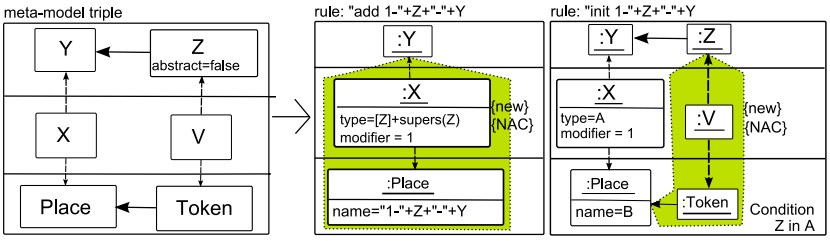

Fig. 24. Constructing the Rules for Translating the Static Information.

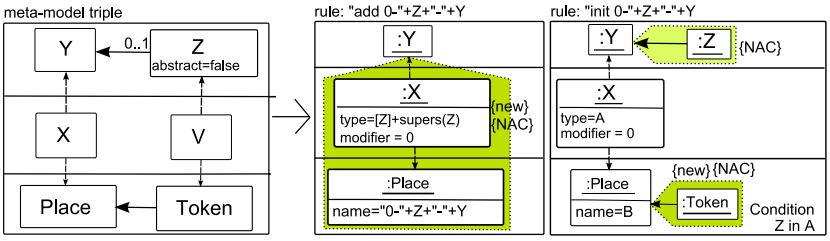

Fig. 25. Constructing the Rules for Translating the Static Information: "0..1" multiplicity.

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.

Additional rules (similar to "add 0-op-machine" and "init 0-op-machine" in Fig. 8) 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 former). The approach for their generation is shown in Fig. 25.

In general, the previous procedure can be generalized for a "0..n" multiplicity, with n some natural number. The meta-model pattern and some of the generated rules are shown in Fig. 26. The idea is that, if the Y object is not connected to any Z object, then n tokens are created in the zero-testing place. If exactly one Z object is connected, then $n-1$ tokens are created and so on. This way, n TGG rules are created, checking the existence of $\{0, 1, .., n-1\}$ connected Z objects. The NACs in each TGG rule ensure that no more than x objects are found. Note that, when testing for 0 objects of type Z, the transitions use the zero testing place with a self loop in which each arc has a weight of n.

Finally, if the multiplicity is "m..n" instead of "0..n", the procedure is similar to the one in Fig. 26, but the rules testing for less than m connected Z objects are not generated.

10.2. TGG Rules for the Dynamic Behaviour

In addition, a TGG rule is constructed for each rule of the source DSVL. As stated before, rules should satisfy conditions R1-R4, in particular rules may have an arbitrary number of NACs, but each with at most one kind of token-like element. The construction algorithm proceeds as follows:

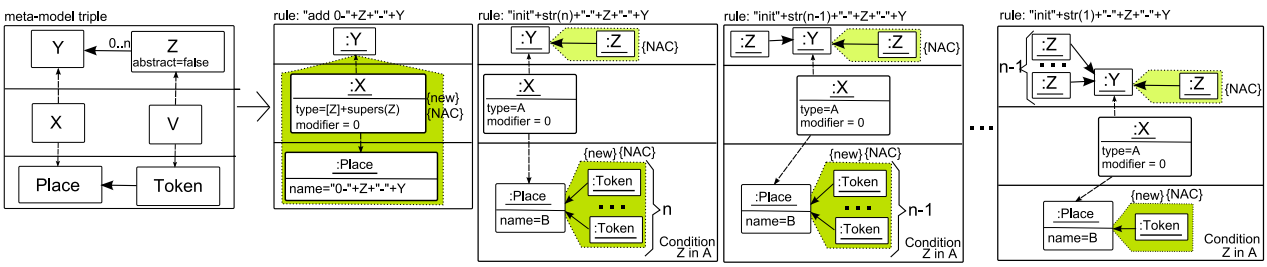

Fig. 26. Constructing the Rules for Translating the Static Information: "0..n" multiplicity.

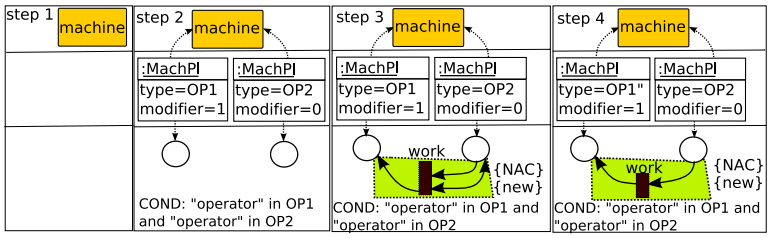

Fig. 27. Steps for Deriving the TGG rule from Rule "work".

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* or untagged. Fig. 27 shows this first step for rule "work" (shown in Fig. 16).
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 the attribute "type" of the corresponding mapping object. For each bounded token-like entity marked *NAC*, *del* or *new*, add an additional mapping identifying the associated zero-testing place. Do not add a mapping twice to the same place. In Fig. 27 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. Add a weight equal to $k - n + 1$ to both arcs in the loop, where k is the upper bound in the meta-model and n the number of token-like elements of the given type in the NAC. 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. 27, 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*. The weight of these two arcs is one, which is the upper bound of the cardinality in the meta-model. 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. The arc from the zero-testing place serves two purposes: it checks that there is room for one operator, and then removes one token, because one operator is added.
4. Simplify connections to/from the Petri net transition to zero-testing places. An incoming edge can be cancelled with an outgoing one, but the testing arcs from zero-testing places arising from the NACs cannot be eliminated. That is, if a NAC states that n elements have to be tested in a zero-testing place, we cannot simplify to obtain a weight smaller than n in the arc from the place to the transition. This is to allow rewriting of token-like entities by a single rule (or just test their presence), 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 and copy the NAC of the place-like elements in the TGG rules (see rule "create move-2" in Fig. 11).

10.3. Preservation of Semantics

In this section, we show that the obtained Petri net reflects the semantics of the original DSVL, in the sense that a rule of the source DSVL is applicable iff some of its associated Petri net transitions are applicable.

Let's first consider the case of a rule without NACs, and later consider the case with NACs. In the

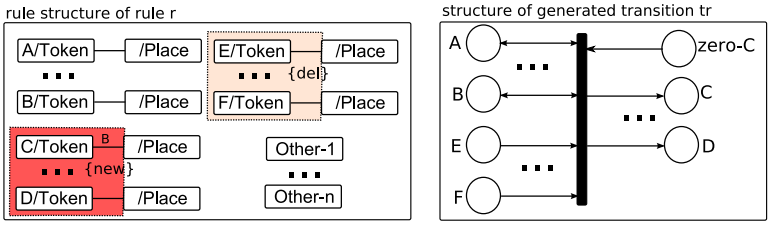

Fig. 28. Structure of a Rule without NACs (left). Generated Transition (right).

general case, a rule can have the structure shown to the left of Fig. 28. We use the notation `A/Token` to denote a token-like element `A`, `/Place` to denote a place-like element and `Other-i` to denote an element with no role (i.e., that is neither place or token-like element). We have decorated with a `B` the edges with bounded cardinality. For simplicity we assume an upper cardinality of one, but the reasoning holds for any cardinality. The rule can have arbitrary edges between the place-like elements and the elements with no role, but they are omitted in the rule scheme. Note that by restriction (R1) of the DSVL rules (see section 5), these can only add or delete token-like elements (together with their edges to place-like elements). For such a rule, a transition with the structure shown to the right of Fig. 28 is created for each occurrence of the place-like elements and elements with no role in the original host graph.

Now we show that (i) if the rule r is applicable at some match, then the generated transition tr for that match is applicable, and both have the same effects and (ii) if some tr is applicable, then its associated rule is applicable to the corresponding match and both have the same effects.

(i) Assume that r is applicable at some match in the original graph. Then, as the place-like elements and elements with no role are static (by restriction R1 no rule can add or delete them), there is some transition tr connected to the associated places of such elements. If the rule is applicable, it means that the token-like elements $A, ..., B, E, ..., F$ are present in the graph, and that (by the implicit post-conditions of the meta-model cardinality constraints) we can create the bounded token-like element C. But note that we translated the existing token-like elements in the original graph to tokens in the corresponding places, and added tokens in the zero-testing places of bounded elements. Hence, if C can be created, this is because there is at least one token in the zero-testing place of C. If applied, the rule creates elements $C, ..., D$ associated to the corresponding place-like elements, deletes $E, .., F$ and preserves $A, ..., B$. Similarly, the transition puts tokens in the places associated to the types of $C, ..., D$, preserves the tokens in $A, ..., B$ and deletes the tokens in $E, ..., F$, and the zero-testing place associated with C. Note that this correctness relation holds for the initial graph, and hence it holds for subsequent graphs derived by rule applications, as the rule and the transition effects are equivalent.

(ii) Assume that some tr is enabled in the initial marking. This means that its in-places contain enough tokens. Hence, the DSVL rule r (from which tr was derived from) is applicable to a match made of: (i) the place-like elements that tr connects and (ii) the token-like elements associated to the places. This is so, because the place-like elements and the elements with no role are never modified, and because initially, we generated one token in the initial marking for each token-like element in the original host graph. Moreover, the rule cannot fail to be applicable due to the dangling condition, because the restriction R2 ensures that token-like elements are deleted together with their respective connections. Also, there is one token in the zero-testing place of C, which means that there is room for another C, and hence the rule cannot fail due to the implicit cardinality constraint post-conditions. Finally, by the same reasons stated in the previous item, the effects of firing the transition are the same as applying the rule. Again, this correctness relation holds for the initial marking, and hence holds for subsequent markings obtained by transition firings, as the transition and the rule effects are equivalent.

Now let's consider a rule with NACs. The general scheme of such rule is shown to the left of Fig. 29. Note that, according to constraints R3 and R4 in section 5, our translation can only handle restricted kinds of NACs. In particular, R3 demands that, if a token-like element is present in a NAC, then its association with a place-like element should be bounded. This ensures that a zero-testing place can be created. In the reasoning we assume an upper bound of one, but it is clear that the reasoning is general enough for other bounded cardinalities. R4 forbids NACs with two or more types of token-like elements. Again, a rule can

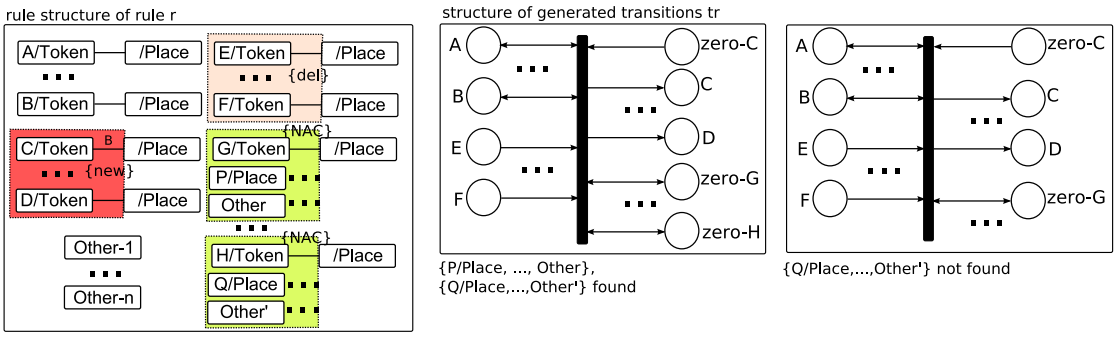

Fig. 29. Structure of a Rule with NACs (left). Generated Transition (right).

have arbitrary edges between the place-like elements and the elements with no role, but they are omitted in the rule scheme.

The right of Fig. 29 shows the structure of the generated transitions. As before, one such transition is created for each occurrence in the initial host graph of the place-like elements and the elements with no role appearing in the rule's LHS. Moreover, as NACs may contain both place-like and one type of token-like element, different transitions are generated. If all the place-like elements in a NAC are found in the graph, we add a loop connection from the zero testing place to check that the token-like element is not present. On the contrary, if there is no match for the place-like elements in the NAC, the zero testing place is not needed, as the NAC will be satisfied at that match.

As in the previous case, we now show that (i) If the rule r is applicable at some match, then the generated transition tr for that match is applicable, and both have the same effects and (ii) If some tr is applicable, then its associated rule is applicable to the corresponding match and both have the same effects.

(i) Assume that r is applicable at some match in the original graph. Then, the argumentation about the elements in the rule's LHS is the same as in the case without NACs. Thus, we only need to show that if each NAC is satisfied, then some of the generated transitions are enabled. A NAC is satisfied either because no match for the place-like elements is found or because it is found, but the token-like elements are not. In the former case, no zero-testing place is connected to the transition, and hence the transition is enabled. In the latter case, the corresponding zero-testing place is linked with the transition. If the token-like element is absent, then the zero-testing place contains one token, as it was added by the TGGs form the original graph, and hence the transition is applicable. Note that as each NAC has at most one token-like element, one token in the zero-testing place is the unique condition to make the NAC satisfied. As in the case without NACs, it can be seen that both the rule and the transition have equivalent effects. Moreover, as each input arc of a place has a correspondence with an output arc of its associated zero-testing place and vice-versa (following the well-known procedure for ensuring capacity constraints in Petri nets [Pet81]), then we can conclude that this correctness relation holds for the initial graph, and therefore for subsequent graphs derived by rule derivations.

(ii) Assume that some tr is enabled in the initial marking. The reasoning about the places which are not zero-testing is as in the previous case. Here we show that if the transition is enabled, then the NACs of the rule are satisfied. This is so because, if the zero-testing places that are present in the transition have one token each, the corresponding NACs are satisfied, because the token-like element in each NAC is absent (and each NAC has at most one type of token-like element). If the rule has more NACs, but no zero-testing place is present in the transition, this means that the NAC is satisfied, because there is no match for the place-like elements in the NAC. Again, it is easy to see that the effects of the rule and the transition are equivalent. As before, we can conclude that this correctness relation holds for the initial marking, and therefore for each reachable marking.

Finally, we have to show that the simplification of arcs for the zero-testing places is correct. Fig. 30 shows the structure of a rule without NACs. The rule adds and deletes elements of the same kind attached to the same place-like element, which is the situation that may give rise to simplification of arcs. There are three possibilities. If more elements are added than deleted, then elements should be deleted from the zero-testing places, as less room is available for new A elements. Note that we need to simplify, otherwise we would be

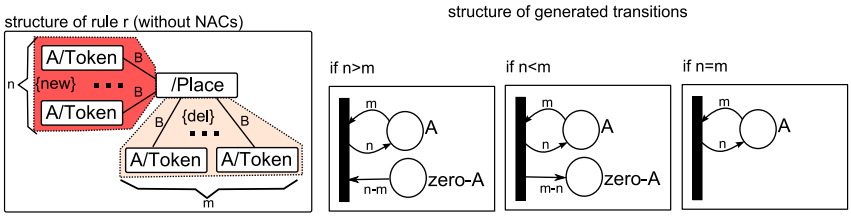

Fig. 30. Simplification of Weight for Zero-Testing Places.

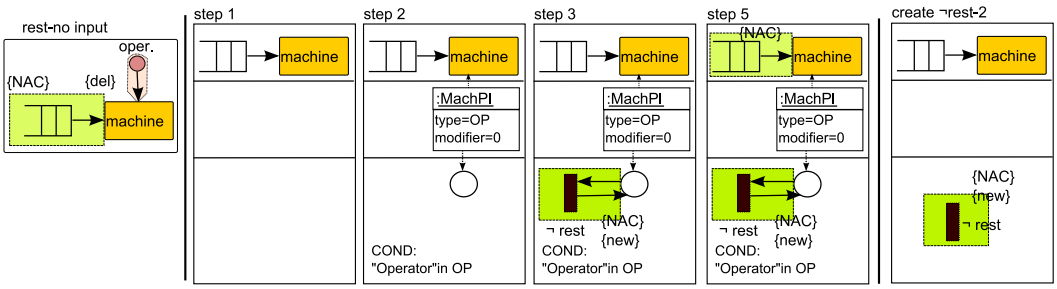

Fig. 31. Generation of Rules for Testing Non-Applicability.

testing the existence of n tokens in the zero-testing place (by taking n tokens, and then putting m back) which is not correct, as we would be demanding n free spaces for for A elements (instead of $n - m$). As in total $n - m$ new elements are added, these are removed from $zero - A$. The second situation arises when more elements are deleted than added. In this case, we have to put tokens in the zero-testing places, as more room is available for new A elements. Finally, if we remove and add the same number of elements, no change is made to the zero-testing place.

If the rule has a NAC, then the simplification works as in the previous cases, but we cannot simplify the weight of the zero-testing place to the transition to become smaller than $k - n + 1$, where k is the upper bound and n is the number of token-like elements in the NAC. This is to make sure that less than n elements are attached to the place-like element (so the NAC is satisfied), but to allow adding and removing elements of the same type as the elements in the NAC.

10.4. TGG Rules for Testing Non-Executability

These rules generate transitions that are enabled if the original rule is not applicable at some match. As an example we use the rule "rest-no input" shown to the left of Fig. 31. The rule is applicable if there is an operated machine without input conveyors. Hence, it is not applicable if some of the following three conditions hold: (i) there is no machine, (ii) the machine is not operated, (iii) the machine has an input conveyor. The procedure for the construction is similar to the one shown in section 10.2 and goes as follows:

1. The first step is the same as the one in section 10.2. In the example, we add the machine and the conveyor, as the former is untagged and the latter belongs to a NAC.
2. This step is like the one in section 10.2, but considering token-like elements labelled NAC, del or untagged, and then adding the correspondig places and zero-testing places. The former are added for NAC token-like elements and the latter for untagged or deleted elements. In the example, the only token-like entity is the operator, which is deleted, and thus we add the zero-testing place associated to the machine.
3. Now, we neglect each unbounded element tagged *new*. For each bounded element tagged *new*, we create a self-loop to its corresponding place with weight $k-n+1$, where k is the upper bound, and n is the number of token-like elements (of the same type) added. This will enable the transition if there is no room to add the n elements. Then, for each element tagged *del* or not marked, we create a self-loop (with weight the number of token-like elements) from its associated zero-testing place to the Petri net transition. This enables the transition if the elements are not present. Finally, for each element marked *NAC*, we create a self-loop (with the number of token-like elements as weight) from its related place to the transition (to

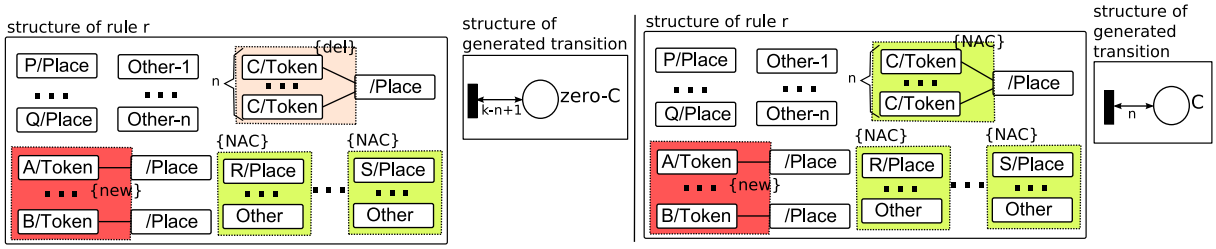

Fig. 32. Structure of Rules for Negative Test and Generated Transitions.

enable the transition if the NAC is not satisfied). In the example, as the operator is deleted, then a loop is created from its zero-testing place. As the transition has self-loops only, it is effect-free.

4. No simplification of arcs is necessary, but if elements of the same type are marked *new* and *NAC*, we take the minimum weight in the self-loop (see the right of Fig. 33).
5. We copy the NACs of place-like elements from the DSVL rule to the TGG. In the example, the conveyor and the arc are labelled as NAC.

The synthesized TGG rule creates the transition only if there is a machine not connected to a conveyor (hence it only checks condition (ii)). This is so, because if this happens, a transition executing "rest" was created at that match. Hence, the "¬rest" transition is enabled if "rest" is not enabled at that match. A TGG rule like "create ¬rest-2" is not needed (creating a transition testing condition (iii)), because, if the machine has no conveyor, then no "rest" transition was generated at that match. Similarly, a transition checking condition (i) (that there is no machine) is not needed either.

Now, we informally show the correctness of the semantics by checking that: (i) the original rule is not applicable at some match iff the generated transition for the given match is enabled and (ii) the transition generated at a match is enabled iff the original rule is not applicable at that match. Recall that the rules we can handle are more restrictive, and in particular they should fulfill the restrictions R5-R8 in section 5.2.

Fig. 32 shows to the left the structure of a rule where `n` token-like elements are deleted. This rule is applicable at a match if (i) all place-like elements and elements with no roles in the LHS are found, (ii) the NACs are satisfied, and (iii) the `n` token-like elements are found. The TGG rule creates the transition shown in Fig. 32 at each occurrence of the place-like elements of the LHS and where the NACs are satisfied. The transition checks the existence of $k-n+1$ tokens in `zero-C`, which exist only if less than `n` token-like entities of type C are present. Hence, if the rule is applicable at that match, the transition is not enabled, and if the rule is not applicable at that match, the transition is enabled (thus (i) holds). Conversely, if the transition is enabled, then the rule is not applicable at that match and vice versa, hence (ii) holds.

The right of the same figure shows another possible rule, where the token-like elements are in a NAC instead of in the LHS. It is easy to see that the rule is applicable at a match where all place-like elements are found and the NACs involving place-like elements are satisfied if there are less than `n` token-like entities of type C. Hence, the generated transition has to check the existence of `n` C entities.

Finally, Fig. 33 shows the case of rules creating bounded token-like elements. In the rule to the left, `n` elements of type A are created, where the maximum bound is $K \geq n$. In such case, the rule fails to be applicable if there are already $k-n+1$ elements of type A connected to the place-like element. This is exactly what the transition that is generated tests. The rule to the right shows the case where some bounded elements are created, and are also part of a NAC. In this case, the rule fails either because there is no room to create `n` elements, or because `m` or more elements already exists. Both conditions can be tested with a self-loop, which checks the most restrictive case.

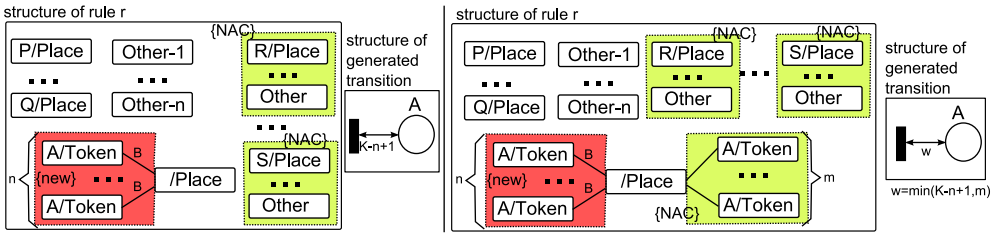

Fig. 33. Structure of Rules for Negative Test and Generated Transitions.