



Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets

J. de Lara^{a,1}, C. Ermel^{b,2}, G. Taentzer^{b,3}, K. Ehrig^{b,4}

^a *Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
Madrid, Spain*

^b *Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik
Berlin, Germany*

Abstract

This work discusses the use of parallel graph transformation systems for (multi-formalism) modeling and simulation and their implementation in the meta-modeling tool ATOM³. As an example, a simulator for Timed Transition Petri Nets (TTPN) is modeled using parallel graph transformation.

Keywords: model simulation, parallel graph transformation, timed transition Petri nets

1 Introduction

The modeling of complex systems has to take into account components of very different nature. These components should be described using different formalisms, each adequate for the view to be described. The problem is to analyze and/or simulate such heterogeneous models. *Multi-formalism modeling*

¹ Juan.Lara@ii.uam.es,

² lieske@cs.tu-berlin.de,

³ gabi@cs.tu-berlin.de,

⁴ karstene@cs.tu-berlin.de

allows software developers to model each component of the system using the most appropriate formalism and solves the simulation problem by identifying a single formalism (called semantic formalism) into which each component is symbolically transferred. Thus, properties of the whole system can be verified if the semantic formalism allows for appropriate formal analysis and simulation techniques. A good candidate for such a semantic formalism are Petri nets as they are equipped by a formal semantics and allow for a formal analysis as well as for the visual simulation of system behavior.

As modeling formalisms today are usually based on diagrams or graphs (e.g. UML diagram types for software engineering or function block diagrams for designing programmable controllers), our approach uses graph transformation [15] for both the transformation of different formalisms to the semantic formalism and the simulation of the transformed model. The different formalisms are defined as visual languages by means of meta-modeling.

In the tool environment AToM³ for multi-formalism modeling and meta-modeling [3], such a meta-model consists of a visual part (class diagrams modeling the symbols used in the visual language) and textual constraints (well-formedness rules in OCL) to restrict the set of models to the meaningful ones. AToM³ allows the generation of model editors and simulators for models of the specified visual language [3], and the transformation of models between formalisms based on different meta-models and graph transformation rules [4]. In this paper, the focus lies on the simulation of a special variant of Petri nets – Timed Transition Petri nets (TTPN) – based on parallel graph transformation [14]. The feature of parallel graph transformation has recently been implemented in AToM³ and allows flexible simulation specifications.

The essence of parallel graph transformation is that (possibly infinite) sets of rules which have a certain regularity, so-called rule schemes, can be described by a finite set of rules modeling the elementary actions. For instance, when modeling the firing of a Petri net transition, the elementary actions would be the removal of a token from a place in the transition's predomain and the addition of a token to a postdomain place. For the description of such rule schemes the concept of amalgamating rules at subrules is used. Thus, the firing of Petri net transitions can be described in a general way although it is not fixed how many input or output places a transition has. This is especially important for AToM³ because here, simulation grammars are defined based on a meta-model, i.e. independent of specific models and can be applied to all valid models (e.g. Petri nets). Thus, the approach of defining Petri net behavior by parallel graph transformation differs significantly from the usual way to map Petri nets to graph grammars [2], where a specific net is considered whose tokens are mapped to discrete nodes labelled by places and where each

transition is mapped to a graph rule.

Other synchronization mechanisms for rule applications are discussed in works on modeling distributed systems based on graph rewriting like [6] (the motivation to develop the amalgamation concept in [14]) and the related, more recent approach of Synchronized Hyperedge Replacement Systems [11] for system reconfiguration.

There are some related tool-based approaches supporting the meta-model based definition of visual languages and the graph-grammar based simulation of visual models, e.g. DiaGen [5] or GenGED [9], but none of them realizes parallel graph transformation. Another related approach is MetaEnv [1], an environment to define the semantics for visual models. A model is translated from an external CASE tool to the semantic domain of high-level timed Petri nets using graph grammars. The semantic model can be simulated and translated into C code. Compared to the AToM³ approach, the semantic model is fixed in MetaEnv, whereas it is variable in AToM³.

Section 2 reviews the concepts of parallel graph transformation. In Section 3, a simulator for Timed Transition Petri nets is presented as a parallel graph transformation system. Section 4 sketches the extensions of AToM³ in order to provide tool support for the definition and application of amalgamated rules.

2 Parallel Graph Transformation

Parallel graph transformation models parallel state transition where a state is described by a graph which can be changed by several actions executed in parallel. Since the graph transformation is rule-based without restrictive execution prescription, it offers the possibility for massively parallel execution. The synchronization of parallel rule applications is described by common subrules. In this section, we review the main concepts of parallel graph transformation [14] and show how the firing of transitions in arbitrary place-transition nets (P/T nets from now on) can be modelled in a concise way using parallel graph grammars. This provides the basis for the simulation of Timed-Transition Petri nets, an extension of P/T nets, discussed in Section 3.

Using graph grammars as modeling technique, objects are represented by nodes and their interrelations by edges. Actions are usually described by graph rules and simulated by graph transformation (which is the application of rules to a given graph, resulting in a new graph modeling the changed system state).

A *graph rule* $r = L \leftarrow K \rightarrow R$ in the double-pushout (DPO) approach to graph transformation [15] consists of three attributed graphs L , K and R and graph morphisms between them (visualized by equal numbers for mapped graph objects in L and R). L and R are the left and right-hand side graphs

and K is their common interface. As a drawing convention, we omit K . All objects with equal numbers in L and R are also contained in K and are preserved when the rule is applied. A rule r can contain one or more negative application conditions (NACs) denoting situations which must not exist for the rule to be applicable. Formally this is expressed by attributed graphs NAC_i and morphisms $n_i : NAC_i \leftarrow L$. A rule is *applicable* to a graph G at a match $m : L \rightarrow G$ if there is no graph morphism $n'_i : NAC_i \rightarrow G$ such that $m = n'_i \circ n_i$ for all $i \in I$. The application of rule r to graph G leads to the derivation of a graph H . Formally, a *derivation* $G \xRightarrow{r} H$ is a DPO construction in the category of attributed graphs and graph morphisms.

The simplest type of parallel actions is that of *independent actions*. If they operate on different objects they can clearly be executed in parallel. If they overlap just in reading actions on common objects, the situation does not change essentially. In graph transformation, this is reflected by a *parallel rule* which is a disjoint union of rules. The overlapping part, i.e. the objects which occur in the match of more than one rule, is handled implicitly by the match of the parallel rule. As the application of a parallel rule can model the parallel execution of independent actions only, it is equivalent to the application of the original rules in either order.

If actions are not independent of each other, they can still be applied in parallel if they can be synchronized by subactions. If two actions contain the deletion or the creation of the same node, this operation can be encapsulated in a separate action which is a common subaction of the original ones. A common subaction is modelled by the application of a *subrule* of all original rules (called *elementary rules*). The application of rules synchronized by subrules is then performed by gluing the elementary rules at their subrules which leads to the corresponding *amalgamated rule*. The application of such a rule is called *amalgamated graph transformation*. An example of an amalgamated rule is given in Fig. 1. Here, two elementary rules *er1* and *er2* have the addition of a loop to a node in common. This common interface is modelled as a subrule. The amalgamated rule contains the common action and, additionally, all actions from the elementary rules that do not overlap. Dashed arrows in Fig. 1 indicate rule morphisms, the vertical arrows are the embeddings of the subrule into the corresponding instances of the elementary rules, and the embeddings of the elementary rules into the amalgamated rule.

Note that there may be arbitrary many instances of each elementary rule which are all glued at one subrule. The number of instances depends on the match of the rule scheme into a specific graph.

Formally, the synchronization possibilities of actions are defined by an interaction scheme. Note that the formal construction of amalgamated rules

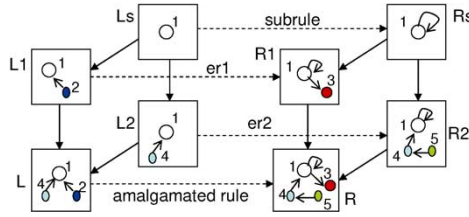


Fig. 1. Construction of an amalgamated graph rule

is described for the DPO approach in [14]. For the extension of this formal basis to attributed graph transformation in the single pushout approach (SPO) with rules enhanced by negative application conditions compare [10].

Definition 2.1 (Interaction Scheme)

An *interaction scheme* $IS = (E, S, SE)$ consists of a set E of elementary rules, a set S of subrules and a set SE of subrule embeddings $se : s \rightarrow e$ into the elementary rules, where $s \in S$ and $e \in E$.

Subrules with NACs can be embedded only into elementary rules containing at least the same NACs as the subrule. In addition, each elementary rule may have other NACs than the subrule. An interaction scheme can also be seen as a bipartite graph (IS-graph) whose nodes are labelled by elementary rules and subrules. Edges are labelled by subrule embeddings.

In addition to the specification how elementary rules should be synchronized, we have to decide where and how often a set of elementary rules should be applied. The basic way to synchronize complex parallel operations is to allow that a rule should be applied at *all different matches* in a given graph. This expresses massively parallel execution of actions. For the purpose in this paper we restrict the *covering of G* (the image of all different matches from instances of elementary rules in G) to all different elementary matches that overlap in the match of their common subrule. For other, more complex covering constructions see [14]. For our restricted covering of G , the IS-graph is always connected.

A graph H is *directly parallel derivable* from graph G by an interaction scheme $IS = (E, S, SE)$ if there is an amalgamated rule r constructed by gluing instances of elementary rules $e_i : L_{e_i} \rightarrow R_{e_i} \in E$ at one of their corresponding subrules $s \in S$ as defined by SE , and if all matches $m_i : L_{e_i} \rightarrow G$ overlap in their subrule match $m_s : L_s \rightarrow G$ such that $G \xrightarrow{r} H$. This rule amalgamation can also be seen as a bipartite graph G_r whose nodes are subrules and instances of elementary rules, and whose edges are subrule embeddings. Each G_r representing a valid rule amalgamation is typed over the IS-graph, e.g. there is a homomorphic mapping from G_r into the IS-graph.

Definition 2.2 (Parallel Graph Transformation System)

A *parallel graph transformation system* $PGTS = (G_0, I)$ consists of a start graph G_0 and a set I of interaction schemes. The language $L(PGTS)$ of a parallel graph transformation system is given by all graphs which are parallel derivable from G_0 by a finite number of direct parallel derivation steps applying amalgamated rules constructed from I .

Definition 2.3 (P/T net simulator as PGTS)

We model P/T nets [13] as attributed graphs with two node types (for places and transitions), two edge types (for arcs from places to transitions and vice versa) and attributes of type *Nat* for the number of tokens and the arc weights. As start graph N of our PGTS, an arbitrary P/T net is allowed. The interaction scheme IS_{seq} (shown as graph on the left top of Fig. 2) describes sequential firing of arbitrary transitions. The subrule *trans* glues together instances of the elementary rules *get* and *put* (see rule *trans*, the instances *get*_1 and *get*_2 of rule *get* and instance *put*_1 of rule *put* in Fig. 2).

The P/T net simulator defined as PGTS in Def. 2.3 leads to the construction of valid sequential P/T net firing rules only. This means, for each transition $t \in N$ in a P/T net N , one amalgamated rule $L \rightarrow R$ is constructed by the simulator PGTS where L and R contain the transition and its environment. In L , markings of predomain places are represented by variables denoting numbers larger than the corresponding arc inscription values. In R , the new markings are computed by subtracting the pre-arc values from the predomain markings and by adding the post-arc values to the postdomain markings.

The subrule *trans* ensures that all copies of elementary rules overlap in the same transition node. The elementary rule *get* removes the number of tokens from a predomain place corresponding to the arc weight, and the rule *put* generates the correct number of tokens on a postdomain place. In each amalgamated firing rule, there are as many copies of *get* and *put* as there are different matches to N , provided that they overlap in the same transition node. Thus, the pre- and postdomain of a transition will be completely covered by the corresponding amalgamated rule. Since the subrule *trans* (and hence all elementary rules) delete the transition and reconstruct it, the dangling condition is not satisfied if not all places from the transition's pre- and postdomain are covered [14]. Fig. 2 shows the construction of an amalgamated rule where the transition is matched to the upper transition of the P/T net N . There are two copies of *get* for the two places in the transition's predomain and one copy of *put* as there is only one place in the postdomain. Note that the variables for token numbers and arc weights are instantiated to different variables in the rule instances.

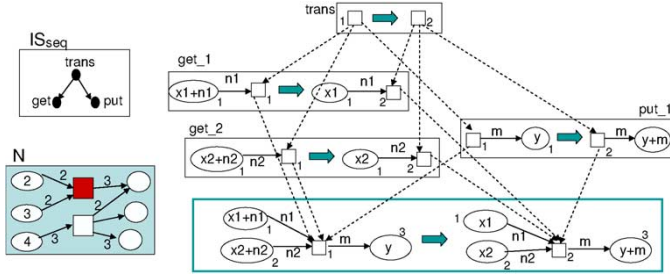


Fig. 2. Firing of transitions in P/T nets

Parallel firing of more than one transition can be similarly expressed by a parallel graph transformation system. The difference is that the elementary rules do not all have to be glued at a single subrule but can be glued at different instances of the subrule. Furthermore, we need two subrules (the transition-gluing one from definition 2.3 and the empty rule) and require a slightly different covering construction (see [14]).

3 A Simulator for Timed Transition Petri Nets

TTPN [12] are like P/T nets, but transitions are assigned a delay, in such a way that they have to be enabled for that amount of time in order to fire. The left of Fig. 3 shows a meta-model for TTPN. Classes *Timer*, *EventQueue* and *Event* are not used in the modeling phase, but in the simulation. During simulation, a unique *Timer* object keeps track of the current simulation time (*current* attribute) and the final time (*final* attribute). Each time a transition is enabled, an event is scheduled and is inserted in the (ordered) event queue (object *EventQueue*). The event is scheduled to occur at the transition firing time. The transition keeps a pointer to the event it has generated by means of relationship *fires*. Once the transition fires, the event is removed from the queue and the simulation time is advanced to the time of the event. This event-driven simulation is a standard technique in discrete event simulation and avoids incrementing time when there is no system state change [8].

The right of Fig. 3 shows a simple model, being an instance of the previous meta-model. It is a queueing model, in which tokens arrive at inter-arrival times of 6, and are served by one of the two parallel servers at rates 7 and 8. The simulation current (0) and final time (100) are shown in the “wall clock” icon, in the upper-left corner. The event queue is shown at the bottom, where the first simulation event is indeed the second one (scheduled at 6), which receives a pointer from the transition that produced it. The event scheduled at -1 is kept in order to make the rules for the simulator simpler. The last

event in the queue is scheduled at the simulation final time, in such a way that the current simulation time can never reach this point.

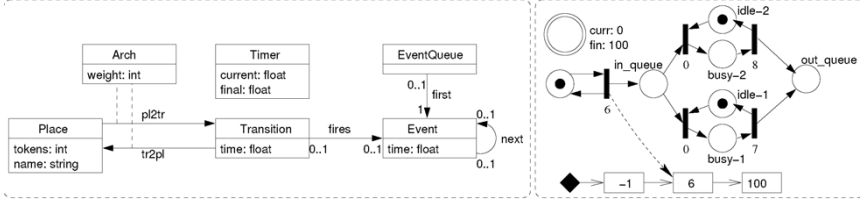


Fig. 3. Meta-model for TTPN (left). TTPN model of a queueing system (right).

In this section we show the specification of a simulator for TTPN using parallel graph transformation systems. We use *atomic firing* (tokens stay in places until transitions fire), *single server* semantics (transitions are provided with just one timer) and *enabling memory* (i.e. timers of transitions being disabled due to a transition firing, are reset). The simulator is composed of two interaction schemes and one regular rule. The first interaction scheme is shown in Fig. 4. Its purpose is to check whether a transition is enabled (which is the case if its input places have at least as many tokens as the weight of the connecting arcs). Then an event is scheduled to occur after the transition enabling time. The scheme is decomposed in two elementary rules glued by one subrule. By convention, we omit in an elementary rule conditions and NACs of the subrule. They are adopted implicitly by each elementary rule.

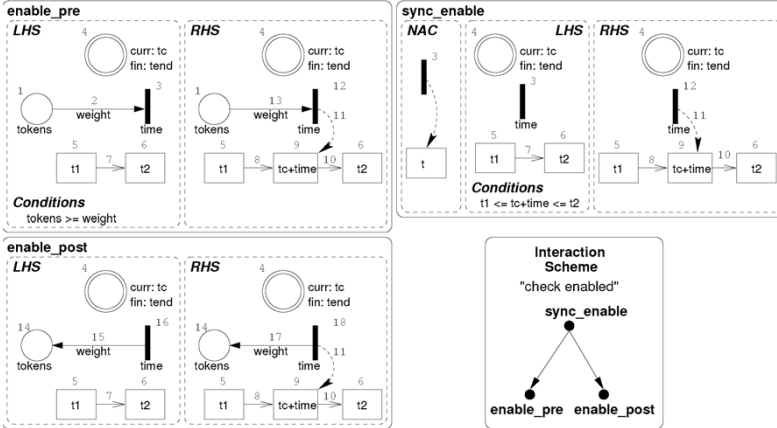


Fig. 4. Check enabled interaction scheme

The *enable_pre* rule can be instantiated once for each predomain place of the transition to be checked. Applying this rule, the transition must not have a scheduled event and the place should have an appropriate number of tokens. The rule also finds the events amongst which the new event will be

placed. The *enable_post* rule is instantiated once for each output place of the transition being checked. The *sync_enable* subrule gives the common context for the instantiation of *enable_pre* and *enable_post*: the transition and the events. If some predomain place of the transition does not have enough tokens, *enable_pre* cannot be instantiated in that place. Analogously to Def. 2.3, the transition is deleted and reconstructed by the subrule, such that in the case of not enough tokens on a predomain place the scheme cannot be applied due to the dangling condition.

Interaction scheme *fire* is shown in Fig. 5 and models a transition firing and the time advance together. Elementary rule *get* is applied to predomain places of the transition whose associated event is the first real event in the queue. The first event (with time equal to -1) and the last event (with time equal to the final simulation time) in the queue are used to simplify the insertion and deletion of events. The time of the first event is the earliest time at which a transition can be fired. Additionally, the current simulation time is advanced to the time of the processed event.

Rule *get* deletes as many tokens as the weight of the connecting arc and erases the event. Rule *put* is similar, but is applied to output places, and thus generates tokens. Subrule *sync_fire* is used to synchronize *put* and *get* by a the common context. Note again that the dangling condition is not met and the scheme cannot be applied, if an incoming place does not have enough tokens. The transition firing time should be less than the final simulation time such that the amalgamation scheme can be applied.

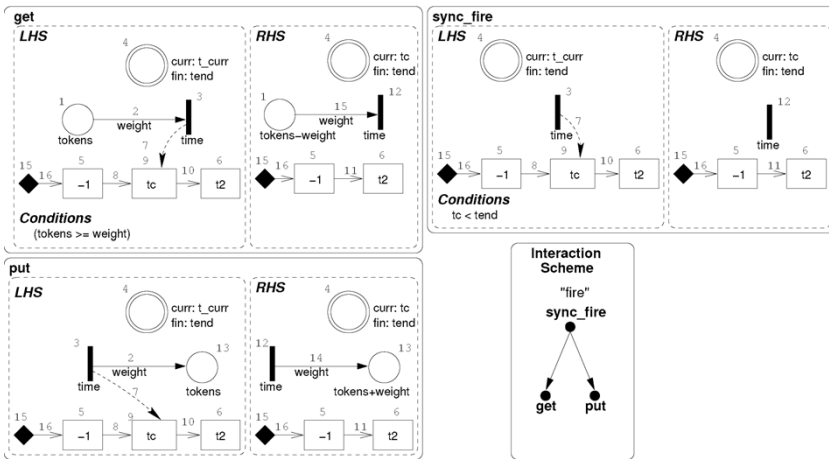
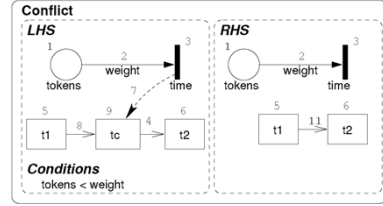


Fig. 5. Fire interaction scheme

In a conflict situation (a place in the pre-domain of two transitions), it is possible for a transition firing to disable other transitions. In this case, the scheduled possible events of the disabled transitions should be erased. This is modelled by rule *conflict* shown on the right.



For the execution, we assign priority 1 to rule *conflict*, priorities 2 and 3 to interaction schemes *check enabled* and *fire* respectively and use the control structure of AToM³, i.e. rules are evaluated in order according to their priority (from lower to higher values). When a rule is applied, the control continues at the rule with the highest priority.

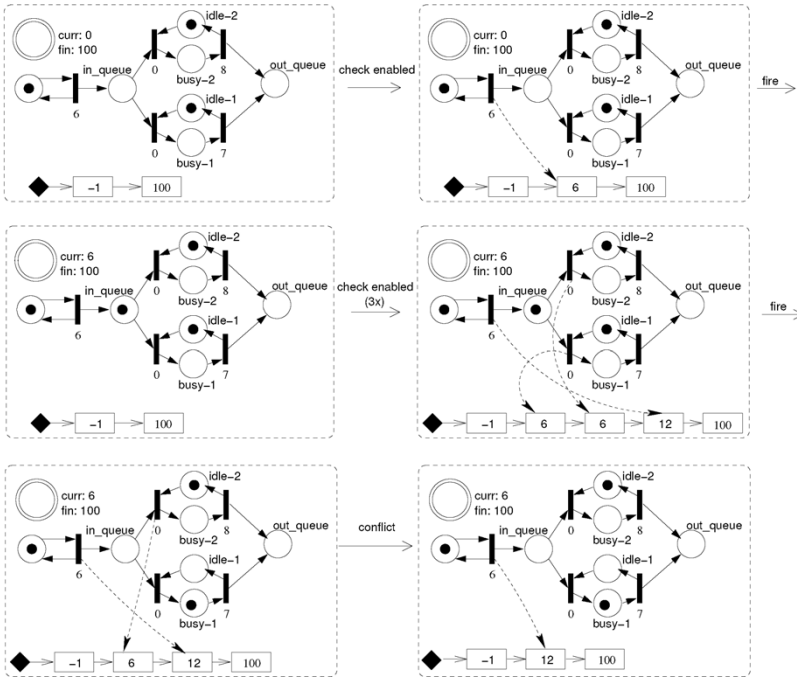


Fig. 6. Derivation sequence of the queueing example

Fig. 6 shows a derivation sequence for the queue example in Fig. 3. In the beginning, the current time *curr* is 0, and the two parallel servers are idle. In order to schedule an arrival, the interaction scheme *check enabled* is applied and time 6 is added to the event queue. The application of interaction scheme *fire* advances the simulation time to 6 and puts a token in the *in_queue* place. In the second row of Fig. 6, three transitions are enabled and their firing times are added to the event queue in the right order. In our case, the events

belonging to the zero-time transitions could have been scheduled in reverse order. Firing the first transition in the event queue leads to a token on place *busy-1*. In the third row of Fig. 6, a conflict situation has to be solved. The upper zero transition is still in the event queue but not enabled anymore. Therefore, the rule *conflict* deletes this event from the queue.

4 Parallel Graph Transformation in AToM³

AToM³ allows defining visual languages by means of meta-models. Model manipulation can be expressed either as Python programs or by means of attributed graph grammars. AToM³'s graph rewriting processor can be configured to work in the single or double pushout approaches.

For the implementation of parallel graph grammars in AToM³, the graph rewriting processor had to be modified only slightly. We took advantage of the meta-modeling capabilities of AToM³ to define a meta-model for amalgamation schemes, and to generate a modeling tool for it. This modeling tool was then incorporated into the AToM³ kernel. Fig. 7 shows AToM³ in the process of modeling the interaction scheme *fire*. A list with the already defined subrules and elementary rules is displayed in the window on the left, while the interaction scheme is shown in the background window (this is the tool obtained by means of meta-modeling). In the front window, the LHS of rule *get* is being edited. Some of the visible attributes are labelled as “ANY” to specify that any value will make a match, while for other attributes a specific value is given. The numbers associated with the nodes and links represent the usual morphisms between the rule left and right hand sides. In addition, rules may have additional conditions (expressed in Python) that must be satisfied in order for the rule to be applied and actions that are triggered if the rule is applied.

Once all the interaction schemes are defined, they can be placed into a list (which may also contain regular rules) and assigned a priority. AToM³ can then execute this list obeying rule priorities. When considering an interaction scheme for application, AToM³ internally builds an amalgamated rule (possibly using several instances of the elementary rules as needed for the current match) and then applies it (if the conditions of all rules are met). Note how, in the Python expressions for the conditions, the user can access attributes of nodes and links by using the number that appears either on the left or right hand sides. As an amalgamated rule can have many instances of an elementary rule, numbers of nodes and links are internally changed if they are not part of any morphism from any subrule to the elementary rule. This implies that the Python expressions for the conditions may have to be changed “on

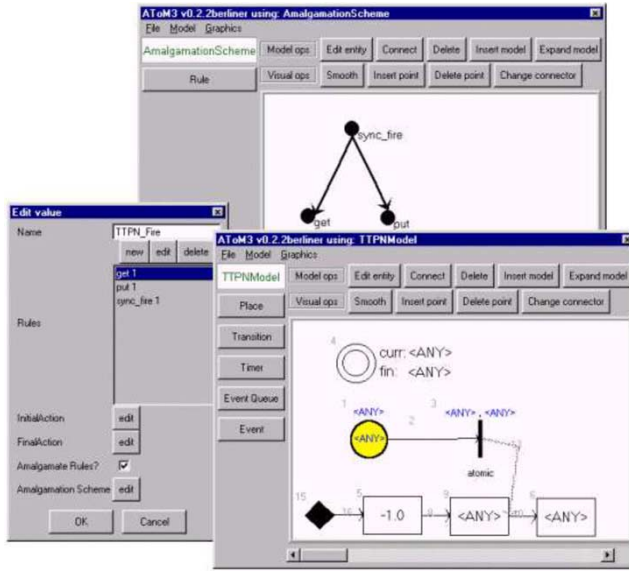


Fig. 7. Definition of a Parallel Graph Transformation System in ATOM³

the fly” once the amalgamated rule is built.

5 Conclusion

This paper has presented parallel graph grammars as a valuable means for modeling and simulation. They extend the capabilities of regular graph grammars by allowing parallel execution of synchronized elementary rules. This is useful for specifying simulators for formalisms in which parallel actions happen, such as Petri nets and others discrete event simulation formalisms. Parallel graph transformation is also useful for formalism transformation where models are translated between different formalisms. In this context, parallel graph transformation simplifies the models transformation as they allow embedding newly created elements in variable contexts. Otherwise, one would have to replicate the transformation rules for each different context. Parallel graph transformation has been implemented in the ATOM³ tool.

In the group at Madrid, parallel and distributed graph grammars are currently applied for modeling simulation protocols in parallel discrete event systems [7]. In the future, we will apply the approach to other TTPN semantics, Petri net variants and discrete event formalisms. The implementation of other covering constructions is also under consideration.

Acknowledgements

This work has been partially sponsored by the SEGRAVIS network and the Spanish Ministry of Science and Technology (TIC2002-01948). The authors also thank the anonymous referees for valuable hints and comments.

References

- [1] Baresi, L., Pezze, M., 2002. *A Toolbox for Automating Visual Software Engineering*, Proc. FASE'02, Springer LNCS 2306, pp. 189–202.
- [2] Corradini, A., Montanari, U., 1995. *Specification of Concurrent Systems: From Petri Nets to Graph Grammars*, Proc. Workshop on Quality of Communication-Based Systems, Berlin, Germany, Kluwer Academic Publishers.
- [3] de Lara, J., Vangheluwe, H., 2002. *AToM³: A Tool for Multi-Formalism Modelling and Meta-Modelling*. In Proc. FASE'02, Springer LNCS 2306, pp. 174 - 188. See also the AToM³ home page, <http://atom3.cs.mcgill.ca>
- [4] de Lara, J. and Taentzer, G., 2004. *Automated Model Transformation and its Validation using AToM³ and AGG*, Proc. Diagrams 2004, Accepted.
- [5] *DiaGen Homepage*, <http://www2-data.informatik.unibw-muenchen.de/DiaGen>
- [6] Degano, P., Montanari, U., 1987. *A Model for Distributed Systems based on Graph Rewriting*. Journal of the ACM, Vol. 34/2, pp. 411–449.
- [7] Ferscha, A., 1995. *Parallel and Distributed Simulation of Discrete Event Systems*. In A. Y. Zomaya, ed., *Parallel and Distributed Computing Handbook*, McGraw Hill, pp. 1003–1041.
- [8] Fishman, G. S., 2001. *Discrete Event Simulation. Modeling, Programming and Analysis*. Springer Series in Operations Research.
- [9] *GenGED Homepage*, <http://tfs.cs.tu-berlin.de/genged>
- [10] Heckel, R., Müller, J., Taentzer, G. and Wagner, A., 1995. *Attributed Graph Transformations with Controlled Application of Rules*. Proc. Coll. on Graph Transformation and its Application in Computer Science, Mallorca, pp 41–53.
- [11] Hirsch, D., 2003. *Graph Transformation Models for Software Architecture Styles*. PhD Thesis, University of Buenos Aires.
- [12] Ramchandani, C., 1973. *Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD Thesis, MIT, Cambridge.
- [13] Reisig, W., 1985. *Petri Nets*. Springer, EATCS Monographs on Theoretical Computer Science, Vol. 4.
- [14] Taentzer, G., 1996. *Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems*. PhD Thesis, Shaker Verlag.
- [15] Taentzer, G., Fischer, I., Koch, M., Volle, V., 1999. *Visual Design of Distributed Systems by Graph Transformation*. Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3., World Scientific. pp 269–340.