Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models[*]

Dániel Varró and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521 Budapest Magyar tudósok körútja 2.
{varro,pataric}@mit.bme.hu

Abstract. As UML 2.0 is evolving into a family of languages with individually specified semantics, there is an increasing need for automated and provenly correct model transformations that (i) assure the integration of local views (different diagrams) of the system into a consistent global view, and, (ii) provide a well–founded mapping from UML models to different semantic domains (Petri nets, Kripke automaton, process algebras, etc.) for formal analysis purposes as foreseen, for instance, in submissions for the OMG RFP for Schedulability, Performance and Time. However, such transformations into different semantic domains typically require the deep understanding of the underlying mathematics, which hinders the use of formal specification techniques in industrial applications. In the paper, we propose a UML-based metamodeling technique with precise static and dynamic semantics (based on a refinement calculus and graph transformation) where the structure and operational semantics of mathematical models can be defined in a UML notation without cumbersome mathematical formulae.

Keywords: metamodeling, formal semantics, refinement, model transformation, graph transformation

1 Introduction

1.1 Evolution of UML

For the recent years UML 1.x has become the de facto standard object-oriented modeling language with a wide range of applications. Its major success is originating in the fact that UML (i) is a *standard* (uniformly understood by different teams of developers) and *visual language* (also meaningful not only for system engineers and programmers but even for customers).

However, based upon academic and industrial experiences, recent surveys (such as [13]) have pinpointed several shortcomings of the language concerning, especially,

[*] This work was partially carried out during the visit of the first author to Computer Science Laboratory at SRI International (333 Ravenswood Ave., Menlo Park, CA, U.S.A.) and supported by the National Science Foundation Grant (CCR-00-86096), the Hungarian Information and Communication Technologies and Applications Grant (IKTA 00065/2000) and the Hungarian National Scientific Foundation Grant (OTKA 038027).

its *imprecise semantics*, and the *lack of flexibility* in domain specific applications. In principle, due to its *in-width* nature, UML would supply the user with every construct he needs for modeling software applications. However, this leads to a complex and hard-to-implement UML language, and since everything cannot be included in UML in practice, it also leads to local standards (profiles) for certain domains.

Recent initiatives for the UML 2.0 RFP (e.g. MML [7]) aim at an *in-depth* evolution of UML into a core kernel language, and an extensible family of distinct languages (called profiles). Each profile (UML diagram) has its own semantics, which architecture fundamentally requires an appropriate and precise metamodeling technique.

1.2 Transformations of UML models

Such a metamodeling-based architecture of UML highly relies on transformations within and between different models and languages. The immense relevance of UML transformations is emphasized, for instance, in submissions to the OMG RFP for a UML sub-language for Schedulability, Performance and Time [16]. In practice, transformations are necessitated for at least the following purposes:

– *model transformations within a language* should control the correctness of consecutive refinement steps during the evolution of the static structure of a model, or define an operational (rule-based) semantics directly on models;
– *model transformations between different languages* should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design;
– a visual UML diagram (i.e., a sentence of a language in the UML family) should be transformed into its (individually defined) semantic domain, which process is called *model interpretation*.

Concerning model transformation in a UML environment, the main course of research is dominated by two basic approaches: (i) transformations specified in (extensions of) UML and OCL [2, 7], and (ii) transformations defined in UML and captured formally by graph transformations [10, 23, 28]. Up to now, OCL-based approaches typically superseded graph transformation-based approaches when considering multilevel static metamodeling aspects. However, when defining dynamic semantics in a visual and operational (if-then-else like) way — where, in fact, the formal technicalities of the underlying mathematics are hidden — graph transformation has clear advantage over OCL.

Our contribution In the paper, we converge the two approaches by providing a *precise* and *multilevel metamodeling framework* where *transformations are* captured visually by a variation of *graph transformations systems*.

1.3 Metamodeling and Mathematics

Previous research (in project HIDE [5]) demonstrated that automated transformations of UML models into various semantic domains (including Petri nets, Kripke automaton, dataflow networks) allow for an early evaluation and analysis of the system. However, preliminary versions of such transformations were rather ad hoc resulting in error prone implementations with an unacceptably high cost (both in time and workload).

In the VIATRA framework [25, 28] (a prototype automated model transformation system), we managed to overcome these problems by providing an *automated methodology for designing transformation* of UML models. After having implemented more than 10 rather complex transformations in this methodology (including model transformations, for instance, for automated program generation [25], formalizing [26] and model checking [14] UML statecharts), we believe that *the crucial step in designing such transformations were to handle uniformly UML and different mathematical domains within the UML framework by metamodeling mathematics*.

Mathematics of metamodeling When regarding the precise semantics of UML (or metamodeling), one may easily find that there is a huge contradiction between *engineering* and *mathematical preciseness*. UML should be simultaneously precise (i) from an engineering point of view to such an extent adequate to engineers who need to implement UML tools but usually lack the proper skills to handle formal mathematics, and, (ii) from a mathematical point of view necessitated by verification tools to reason about the system rigorously.

The UML 2.0 RFP requires (votes for) engineering preciseness: “UML should be defined without complicated mathematical formulae.” However, when considering model interpretations of UML sublanguages (i.e., the abstract syntax of a UML model is mapped into a corresponding semantic domain such as Petri nets, finite automaton, etc.), the proper handling of formal mathematics is indispensable for developing automated analysis tools.

Metamodeling mathematics Meanwhile, recent standardization initiatives (such as PNML [1], GXL [20], GTXL [24], or MathML [29]) aim at developing XML based description formats for exchanging models of mathematical domains between different tools. Frequently (as e.g. in [24]), such a document design is driven by a corresponding UML-based metamodel of the mathematical domain. However, improper metamodeling of mathematics often results in conceptual flaws in the structure of the XML document (e.g., in PNML, arcs may lead between two places, which is forbidden in the definition of Petri nets).

Our contribution Our paper argues that (i) to provide preciseness in UML-based system design, formal mathematical domains should be integrated with UML by transformations, however, (ii) mathematical models can be understood by engineers if they are presented and specified visually by means of metamodels and graph transformation.

The rest of the paper is structured as follows. We first present a refinement calculus for the static parts of multilevel metamodels based on set-theoretic definitions of mathematical structures in Sec. 2. Afterwards, in Sec. 3, a graph transformation-based framework is presented for the specification of dynamic operational semantics of models, including the hierarchic design and reuse of such semantic rules. Finally, Sec. 4 concludes our paper.

2 Structural Refinement of Metamodels

Below we define a structural refinement calculus on set theoretical basis (i.e., refinement of sets, relations and tuples) for a subset of static UML constructs. Our metamodeling framework is *gradually extensible in depth,* thus it only contains a very limited number of core elements, which highly decreases the efforts related to implementation. Moreover, in order to avoid replication of concepts and shallow instantiation (well-known problems of metamodeling identified in [4]) we introduce *dynamic (or fluid) metalevels* where the type–instance relationship is interpreted between models instead of (meta)levels. Our approach has the major advantage that the type–instance relation can be reconfigured dynamically throughout the evolution of models, thus transformations on (traditional) model and metamodel-levels can be handled uniformly.

2.1 Visual Definition of Petri Nets

Before a precise and formal treatment, our goals are summarized informally on a metamodeling example deliberately taken from a well-known mathematical domain, i.e. Petri nets. Petri nets are widely used means to formally capture the dynamic semantics of concurrent systems. However, due to their easy-to-understand visual notation and the wide range of available tools, Petri net tools are also used for simulation purposes even in industrial projects (reported e.g. in [22]). From an UML point of view, transforming UML models to Petri nets provide dependability [6] and performance analysis [12] for the system model in early stages of design.

A possible definition of (the structure of) Petri nets is as follows.

Definition 1. *A Petri net PN is a bipartite graph with distinct node sets P (**places**) and T (**transitions**), edge sets IA (**input arcs**) and OA (**output arcs**), where input arcs are leading from places to transitions, and output arcs are leading from transitions to places. Additionally, each place contains an arbitrary (non-negative) number of **tokens**).*

Now, if we assign a UML class to each set of this definition (thus introducing the **entity** of Place, Transition, InArc, OutArc, and Token), and an association for each allowed connections between nodes and edges (**connections** such as fromPlace, toPlace, fromTrans, toTrans, and tokens), we can easily obtain a *metamodel of Petri Nets* (see the upper right corner of Fig. 1) that seems to be satisfactory.

However, we have not yet considered a crucial part of the previous definition, which states that a Petri net is, in fact, a bipartite graph. For this reason, after looking up a textbook on graph theory, we may construct with the previous analogy the *metamodel of bipartite graphs* (depicted in the lower left corner of Fig. 1) with 'boy' and 'girl' nodes[1], and 'boy-to-girl' and 'girl-to-boy' edges. Moreover, if we focus on the fact that every bipartite graph is a graph, we may independently obtain a *metamodel of graphs* (see the upper left corner of Fig. 1).

In order to inter-relate somehow these metamodels, we would like to express that, for instance, (i) the class Node is a supertype of class Boy, (ii) the association fromPlace

[1] Bipartite graphs are often explained as relations between the set of boys and girls.

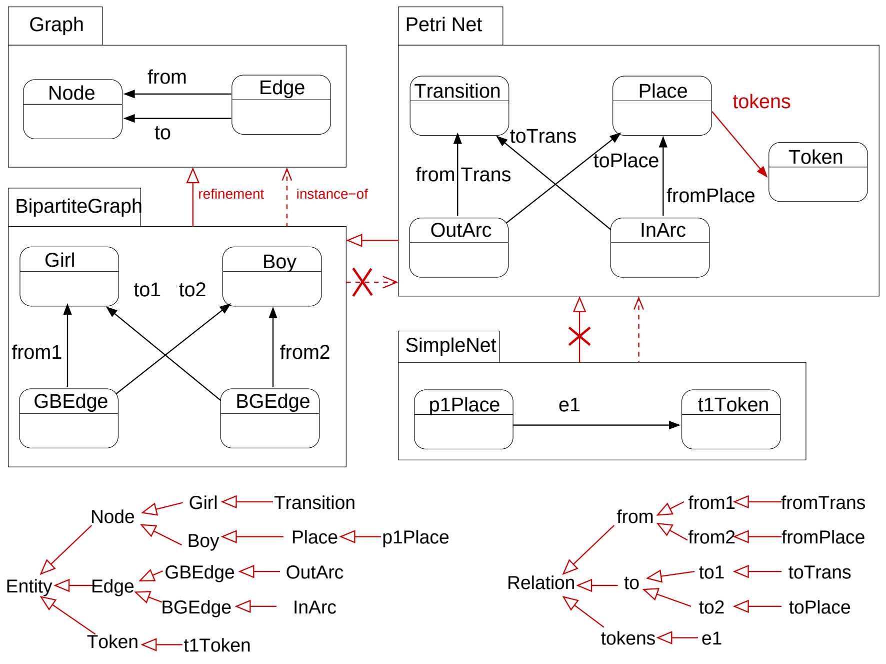

Fig. 1. Defining the structure of Petri Nets

is inherited (indirectly) from the association from, and (iii) the metamodel of bipartite graph is a generalization of the metamodel of Petri nets. In the rest of the paper, we denote these relations uniformly by the term **refinement**, which simultaneously refers to the refinement of entities, connections, and (meta)models.

Our notion of refinement should also handle the instantiations of classes. For instance, in the lower right corner of Fig. 1, a Petri net model SimpleNet consisting of a single place with one token is depicted. This model is regarded as an instance of the Petri net metamodel as indicated by the dashed arrow between the models.

From a practical point of view, supposing that we have an extensible metamodel library, a new metamodel can be derived from existing ones by refinement. Our main goal is to show that (i) mathematical and metamodel constructs can be handled uniformly and precisely, and (ii) the dynamic operational semantics of models can also be inherited and reused with an appropriate model refinement calculus in addition to the static parts of the models.

2.2 Formal semantics of static model refinement

Our metamodeling framework VPM uses a minimal subset of MOF constructs (i.e., classes, associations, attributes, and packages) with a slightly re-defined (better to say, precisely defined) semantics, which has a direct analogy with the basic notions of mathematics, i.e., sets, relations, functions, and tuples (tuples are constituted in turn from sets, relations and other tuples).

However, in order to avoid clashes between notions of MOF and set theory as much as possible, a different naming convention is used in the paper, which simultaneously refers to UML and mathematical elements. A **model element** in VPM may be either an **entity**, a **connection**, a **mapping**, or a **model** (see the MOF metamodel of our approach in Fig. 2). Each of these constructs is indexed by **a unique identifier** (accessed by a $.id$ postfix in the sequel) *and* **a set** including the *identifier of the entity* and the *identifiers of the (intended) refinements of the entity* (accessed by a $.set$ postfix).[2]

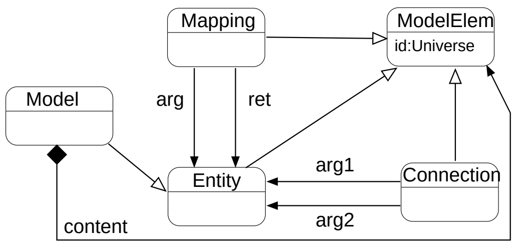

Fig. 2. The MOF metamodel of our approach

– An **entity** E, which is the most basic element of the model space. Entities are represented visually by UML Classes.
– A **connection** R between two entities is a binary relation between the associated sets. Connections are depicted as (directed) associations.
– A **mapping** F from entity E_1 to entity E_2 is a function with the domain of (the set of) E_1 and range of E_2. Mappings can be denoted visually by an attribute assigned to the entity of its domain.
– A **model** (or structure) M is a tuple consisting of the corresponding sets, relations, functions and tuples of entities, connections, mappings, and models, respectively. Models will be represented by UML packages. A model that only consist of a single entity is also regarded as an entity.

The static semantics of our metamodeling framework is based upon a refinement calculus (depicted by UML generalization relations) that handles the traditionally distinct notions of inheritance and instance-of relations uniformly.

1. **Entity refinement**: $E_1 \Rrightarrow E_2 \overset{\text{def}}{=} E_1.set \subseteq E_2.set$ (E_1 is a(n entity) refinement of E_2, alternatively, E_2 is an abstraction of E_1). Informally, E_2 is a superclass of E_1 in the terms of MOF metamodeling.

[2] This philosophy is in analogy with the axiomatic foundations of set theory. There we have classes as a notion that remains undefined. An element of a class is by definition a set, while the singleton class that contains this element is also a set.

2. **Connection refinement**: $R_1(A_1, B_1) \mapsto R_2(A_2, B_2) \overset{\text{def}}{=} R_1.set \subseteq R_2.set \wedge A_1 \mapsto A_2 \wedge B_1 \mapsto B_2$ (where all A_i and B_i are entities). Connection refinement expresses the fact that MOF associations can also be refined during the evolution of metamodels.
3. **Mapping refinement**: $F_1(A_1) : B_1 \mapsto F_2(A_2) : B_2 \overset{\text{def}}{=} F_1.set \subseteq F_2.set \wedge A_1 \mapsto A_2 \wedge B_1 \mapsto B_2$ (i.e., functions are treated as special relations). From a practical point of view, MOF attributes are also considered in our metamodeling framework.
4. **Model refinement**: $M_1^n \mapsto M_2^k \overset{\text{def}}{=} M_1.set \subseteq M_2.set \wedge \forall i \exists j : M_1[j] \mapsto M_2[i]$ (M_1 is a model *refinement* of M_2), where $M[i]$ is the ith argument (component) of tuple M. Informally, there exists a refinement of each argument of M_2 in a corresponding argument of M_1. In MOF terms, each class in the super model is refined into an appropriate class of the submodel. However, this latter one may contain additional classes not having origins in the supermodel.
5. **Model instance**: $M_1^n \mapsto M_2^k \overset{\text{def}}{=} M_1.set \subseteq M_2.set \wedge \forall i \exists j : M_1[i] \mapsto M_2[j]$ (M_1 is a (model) *instance* of M_2). Informally, there exists a *refinement* of each component of M_1 in a corresponding component of M_2. In practice, each object in the instance model has a proper class in the metamodel. However, this latter one may contain additional classes without objects in the instance model.
6. **Instantiation, typing**: a model element $X \in M_1$ is an *instance* of model element $Y \in M_2$ (alternatively, Y is the *type* of X), denoted as , if M_1 is a model instance of M_2, and X is a refinement of Y. Formally, $M_1.X \mapsto M_2.Y \overset{\text{def}}{=} M_1 \mapsto M_2 \wedge X \mapsto Y$. In MOF terms, a class is said to be the *type* of an object, if the corresponding model of the object is an instance of the (meta)model related to the class.

In order to close the refinement calculus from the top, we suppose that there exists a unique top-most model containing a unique top-most entity, connection and function.

A main advantage of our approach (in contrast to e.g. [3]) is that *type-instance relations can be reconfigured dynamically*. On one hand, as a model can take the role of a metamodel (thus being simultaneously a model *and* a metamodel) by altering only the single instance relation between the models, we avoid the problems of "replication of concepts" and "shallow instantiation". On the other hand, transformations on different metalevels can be captured uniformly, which is an extremely important feature when considering the evolution of models through different domains [23].

2.3 Formalizing the Petri net metamodel hierarchy

The theoretic aspects of model refinement (and instantiation) are now demonstrated on the Petri net metamodel hierarchy. Supposing that the refinement relations depicted at the bottom of Fig. 1 hold between the model elements (e.g., Boy is a refinement of Node, e1 is a refinement of tokens; the interested reader can verify that all the connection refinements are valid) we can observe the following.

Proposition 1. *BipartiteGraph is both* a model refinement and instance *of Graph.*

Proof. The proof consist of two steps.

1. **Proof of refinement:** for each element in the Graph model there exist a refinement in the BipartiteGraph. Girl is refinement of Node; GBEdge is of Edge; from1 is derived from from; and to1 is refined from to.
2. **Proof of instantiation:** for each element in Bipartite Graph there exist a refinement in Graph. Girl and Boy are refinements of Node; GBEdge and BGEdge are of Edge; from1 and from2 are of from; and to1 and to2 are of to. □

By similar course of reasoning, we can prove all the other relations between different models of Fig. 1. Note that Petri Net is not an instance of Bipartite Graph (as Token is a new element), and SimpleNet is not a refinement of Petri Net (since, for instance, there are no transitions).

3 Dynamic Refinement in Operational Semantic Rules

Now we focus our attention to extend the static metamodeling framework by a general and precise means for allowing the user to *define* the evolution of his models (dynamic operational semantics) in a hierarchical and operational way. Our approach uses *model transition systems* [28] as the underlying mathematics, which is formally a variant of graph transformation systems enriched with control information.

However, from a UML point of view, model transition systems are merely a pattern-based manipulation of models driven by an activity diagram (specified, in fact, by a UML profile in our VIATRA tool), thus the domain engineer only has to learn a framework that is very close to the concepts of UML. After specifying the semantics of the domain, a single virtual machine of model transition systems may serve a general meta-simulator for arbitrary domains.

In the current section, we first demonstrate that model transition systems are rich enough to be a general purpose framework for specifying dynamic operational semantics of modeling languages in engineering and mathematical domains by constructing (an executable) formal semantics for Petri nets. Afterwards, we define the notion of rule refinement, which allows for a controlled reuse of semantic operations of abstract mathematical models (such as graphs, queues, etc.) in engineering domains.

3.1 An Introduction to Model Transition Systems

Graph transformation (see [19] for theoretical foundations) provides a rule-based manipulation of graphs, which is conceptually similar to the well-known Chomsky grammar rules but using graph patterns instead of textual ones. Formally, a **graph transformation rule** (see e.g. addTokenR in Fig. 3 for demonstration) is a triple $Rule = (Lhs, Neg, Rhs)$, where Lhs is the left-hand side graph, Rhs is the right-hand side graph, while Neg is (an optional) negative application condition (grey areas in figures).

The **application** of a rule to a **model (graph)** M (e.g., a UML model of the user) alters the model by replacing the pattern defined by Lhs with the pattern of the Rhs. This is performed by

1. *finding a match* of the Lhs pattern in model M;

2. *checking the negative application conditions* Neg which prohibits the presence of certain model elements;
3. *removing* a part of the model M that can be mapped to the Lhs pattern but not the Rhs pattern yielding an intermediate model IM;
4. *adding* new elements to the intermediate model IM which exist in the Rhs but cannot be mapped to the Lhs yielding the derived model M'.

In a more operational interpretation, Lhs and Neg of a rule defines the *precondition* while Rhs defines the *postcondition* for a rule application.

In our framework, graph transformation rules serve as elementary operations while the entire operational semantics of a model is defined by a **model transition (transformation) system** [28], where the allowed transformation sequences are constrained by **control flow graph** (CFG) applying a transformation rule in a specific **rule application mode** at each node. A rule can be executed (i) parallelly for all matches as in case **forall** mode; (ii) on a (non-deterministically selected) single matching as in case of **try** mode; or (iii) as long as applicable (in **loop** mode).

3.2 Model transition system semantics of Petri nets

In order to demonstrate the expressiveness of our semantic basis (in addition to the technicalities of graph transformation), we provide a model transition system semantics for Petri nets (in Fig. 3) based on the following definition.

Definition 2 (Informal semantics of Petri nets). *A micro step of a Petri net can be defined as follows.*

1. *A transition is enabled when all the places with an incoming arc to the transition contain at least one token (we suppose that there is at most one arc between a transition and a place).*
2. *A single transition is selected at a time from the enabled ones to be fired.*
3. *When firing a transition, a token is removed from each incoming place, and a token is added to each outgoing place (of a transition).*
4. *When no transitions are enabled the net is dead.*

At the initial step of our formalization, we extend the previous metamodel of Petri nets by additional features (such as mappings/attributes enable and fire, or connections add and del) necessitated to capture the dynamic parts. Then the informal interpretation of the rules (given in the order of their application) is as follows. Note that to improve the clarity, the types of each model element are depicted textually instead of the original graphical representation by inheritance relations. Thus, for instance, T:Trans can be interpreted as T is a direct refinement of Trans.

1. The enable and fire attributes are set to false for each transition of the net by applying rules delEnableR and delFireR in *forall* mode.
2. A transition T becomes enabled (by applying enableTrR) only if for all incoming arc A linked to a place P, this place must contain at least one token K (note the double negation in the negative conditions).

UML 2002: Fifth International Conference on the Unified Modeling Language
pp. 18–33, LNCS 2460, Springer, Sept 30 – Oct 4, 2002, Dresden, Germany

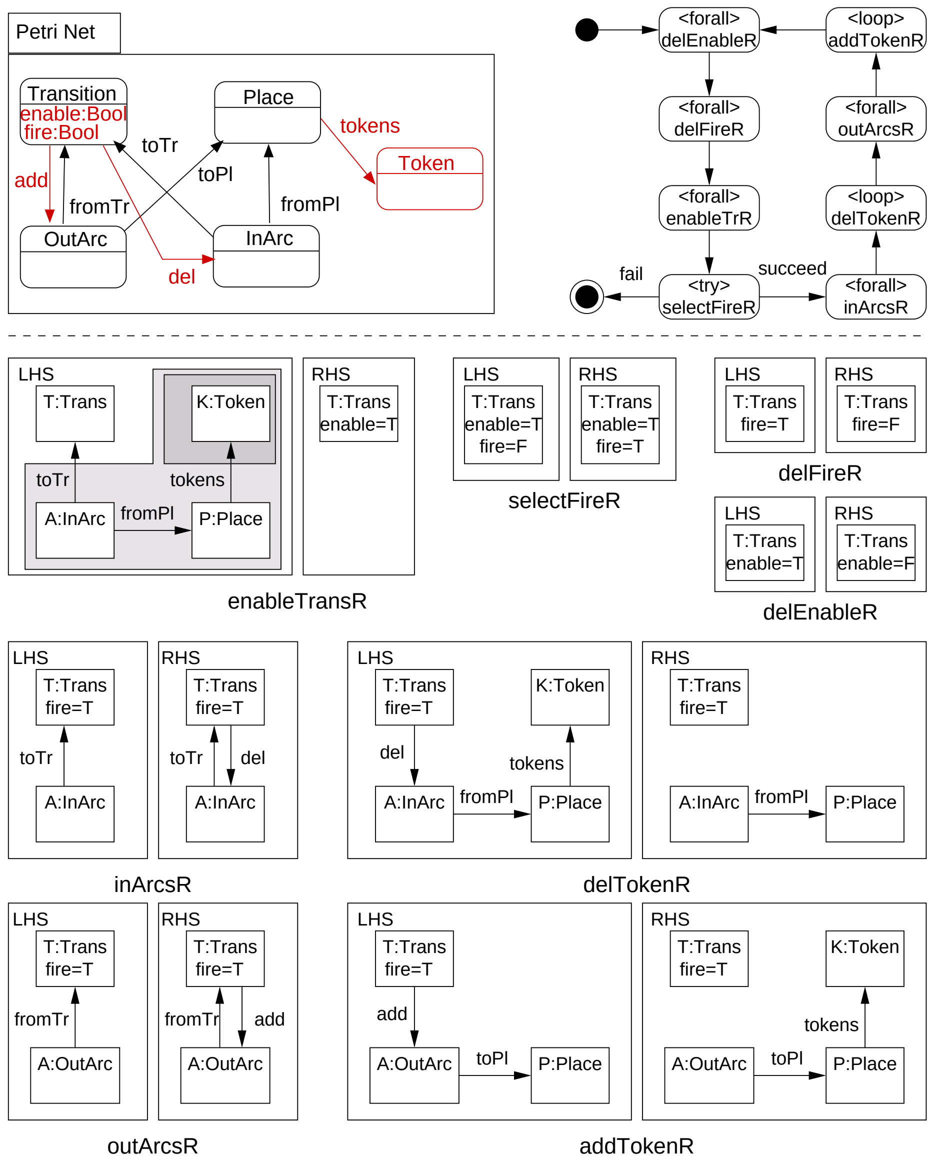

Fig. 3. Model transition semantics of Petri nets

3. A single transition is selected non-deterministically by executing selectFireR in *try* mode. If no transitions are enabled then the simulation of the net is finished.
4. All the InArcs are marked by a del edge that lead to the transition selected to be fired by the application of rule inArcsR.

5. A token is removed from all places that are connected to an InArc marked by a del edge. The corresponding rule (delTokenR) is applied as long as possible (in loop mode) and a del edge is removed in each turn.
6. A process similar to Step 4 and 5 marks the places connected to the transition to be fired by add edges and generates a token for each of them, and the micro step is completed.

3.3 Rule refinement

A main goal of multilevel metamodeling is to allow a hierarchical and modular design of domain models and metamodels where the information gained from a specific domain can be reused (or extended) in future applications. Up to now, metamodeling approaches only dealt with the reuse of the static structure while the reuse of dynamic aspects has not been considered. However, it is a natural requirement (and not merely in mathematical domains) that if a domain is modeled as a graph then all the operations defined in a library of graphs (such as node/edge addition/deletion, shortest path algorithms, depth first search, etc.) should be adapted for this specific domain without further modifications.

Moreover, semantic operations are frequently needed to be organized in a hierarchy (and/or executed accordingly). For instance, in feature/service modeling, we would like to express for the user that an operation copying highlighted text from one document to another and another operation that copying a selected file between two directories are conceptually similar in behavior when regarding from a proper level of abstraction. To capture such semantic abstractions, we define *rule refinement* as a precise extension of metamodeling for dynamic aspects of a domain as follows.

A model pattern M_1 appearing in a graph transformation rule (either in Lhs, Rhs, or Neg) is a **typed subpattern** (or typed subgraph) of a model pattern M_2 (denoted as $M_1 \leq M_2$) if all model elements E_1 of M_1 can be mapped (isomorphically) to a corresponding element E_2 of M_2 such that the type of E_2 is a refinement of the type of E_1. Informally, pattern M_1 is more general than pattern M_2 in the sense that whenever the matching of M_2 succeeds it immediately implies the successful matching of M_1.

Based upon the definition of typed subpatterns, the refinement relation of typed rules is defined as follows: a rule $r_1 = (Lhs_1, Rhs_1, Neg_1)$ is a **refinement** of rule $r_2 = (Lhs_2, Rhs_2, Neg_2)$ (denoted as $r_1 \mapsto r_2$) if

1. $Lhs_2 \leq Lhs_1$: the positive preconditions of r_2 are weaker than of r_1
2. $Neg_2 \leq Neg_1$: the negative preconditions of r_2 are weaker than of Lhs_1. As a result, r_2 can be applied whenever r_1 is applicable
3. $Rhs_2 \setminus Lhs_2 \leq Rhs_1 \setminus Lhs_1$: r_1 adds at least the elements that are added by the application of r_2
4. $Lhs_2 \setminus Rhs_2 \leq Lhs_1 \setminus Rhs_1$: r_1 removes at least the elements that are deleted by the application of r_2. As a result, the postconditions of r_1 are stronger than of r_2.

The concepts of rule refinement are demonstrated on a brief example (see Fig. 4). Let us suppose that a garbage collector removes a Node from the model space (by applying rule delNodeR) if reference counter of the node has been decremented to

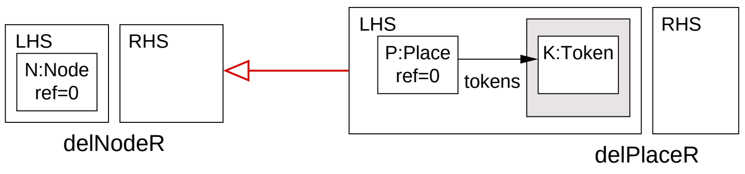

Fig. 4. Rule refinement

0 (denoted by the attribute condition ref=0). Meanwhile, in case of Petri nets, we may forbid the presence of tokens not assigned to a place. Therefore, even when the reference counter of a Place (which used to be a refinement of Node) reaches 0, an additional test is required for the non-existence of tokens attached. If none of such tokens are found then the place P can be safely removed (cf. rule delPlaceR).

Proposition 2. *Rule delPlaceR is a refinement of rule delNodeR. Therefore, in typical applications (visual model editors, etc.) delPlaceR takes precedence of delNodeR.*

Proof. The three steps of the proof are the following:

1. $Lhs_{delNodeR} \leq Lhs_{delPlaceR}$ since Place (the type of P) is a refinement of Node (the type of N).
2. $Neg_{delNodeR} \leq Lhs_{delPlaceR}$ since rule delNodeR has no negative conditions.
3. Conditions 3 and 4 trivially hold for the empty right-hand sides of rules.

4 Conclusions

We presented a visual, and precise metamodeling (VPM) framework that is capable of uniformly handling arbitrary models from engineering and mathematical domains. Our approach based upon a simple refinement calculus provides a multilevel solution covering the visual definition of both static structure and dynamic behavior.

Compared to dominating metamodeling approaches, VPM has the following distinguishing characteristics.

– *VPM is a multilevel metamodeling framework*. The majority of metamodeling approaches (including ones that build upon the MOF standard [17]; GME [15], PROGRES [21] or BOOM [18]) considers only a predefined number of metalevels. While only [3] (a framework for MML) and [4] supports multilevel metamodeling. By the *dynamic reconfiguration of type-instance relationship* between models, VPM provides such a solution that avoids the problem of replication of concepts (from which [3] suffers as identified in [4]).
– *VPM has a visual (UML-based) and mathematically precise specification language for dynamic behavior.* Like [9,21] for UML semantics, or [23] for model migration through different domains, VPM uses a variant of graph transformation systems (introduced in [28]) for capturing the dynamic behavior of models, which provides a purely visual specification technique that fits well to a variety of engineering domains.

– *VPM provides a reusable and hierarchical library of models and operations*. Extending existing static and dynamic metamodeling approaches, models and operations on them are arranged in a hierarchical structure based on a simple refinement calculus that allows a controlled reuse of information in different domains. Initiatives for a reusable and hierarchical *static* metamodeling framework include MML [7] and GME [15], however, none of them provides reusability for rules.
– *VPM supports model transformations within and between metamodels*. The model transformation concepts of VPM is built on results of previous research [27, 28] in the field. Similar applications have been reported recently in [10, 11].
– *VPM can precisely handle mathematical domains* as demonstrated by the running example of the paper.

The theoretical foundations introduced in the paper are supported by a prototype tool called VIATRA (VIsual Automated model TRAnsformations) [28]. VIATRA has been designed and implemented to provide automated transformations from between models defined by a corresponding MOF metamodel (tailored, especially, to transformation from UML to various mathematical domains). Recently, this tool is being extended to support the multilevel aspects of the VPM approach.

Further research is aiming at (i) to design a pattern-based constraint language for expressing static requirements, (ii) to provide model checking facilities for specification based on VPM, and (iii) an automated translation of mathematical structures (from formal definitions given in a MathML format) into their corresponding VPM metamodel.

Acknowledgments We would like to thank Gergely Varró, John Rushby and many of his colleagues at SRI International for their valuable comments.

References

1. *Petri Net Markup Language*. URL `http://www.informatik.hu-berlin.de/top/pnml`.
2. D. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. Ph.D. thesis, University of Kent, Canterbury, 2000.
3. J. Alvarez, A. Evans, and P. Sammut. Mapping between levels in the metamodel architecture. In M. Gogolla and C. Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 34–46. Springer, 2001.
4. C. Atkinson and T. Kühne. The essence of multilevel metamodelling. In M. Gogolla and C. Kobryn (eds.), *Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 19–33. Springer, 2001.
5. A. Bondavalli, M. D. Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.
6. A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analyses for supporting design decisions in UML. In *Proc. HASE'99: The 4th IEEE International Symposium on High Assurance Systems Engineering*, pp. 64–71. 1999.
7. T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation semantics for UML. In H. Hussmann (ed.), *Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy*, vol. 2029 of *LNCS*, pp. 17–31. Springer, 2001.

8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
9. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard*, vol. 1939 of *LNCS*, pp. 323–337. Springer, 2000.
10. G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn (eds.), *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 272–286. Springer, 2001.
11. R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 11–21. Grenoble, France, 2002.
12. G. Huszerl and I. Majzik. Quantitative analysis of dependability critical systems based on UML statechart models. In *HASE 2000, Fifth IEEE International Symposium on High Assurance Systems Engineering*, pp. 83–92. 2000.
13. C. Kobryn. UML 2001: A standardization Odyssey. *Communications of the ACM*, vol. 42(10), 1999.
14. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, vol. 11(6):pp. 637–664, 1999.
15. A. Ledeczi., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proc. Workshop on Intelligent Signal Processing*. 2001.
16. Object Management Group. *UML Profile for Schedulability, Performance and Time*. URL `http://www.omg.org`.
17. Object Management Group. *Meta Object Facility Version 1.3*, 1999. URL `http://www.omg.org`.
18. G. Övergaard. Formal specification of object-oriented meta-modelling. In T. Maibaum (ed.), *Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany*, vol. 1783 of *LNCS*. Springer, 2000.
19. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations.* World Scientific, 1997.
20. A. Schürr, S. E. Sim, R. Holt, and A. Winter. The GXL Graph eXchange Language. URL `http://www.gupro.de/GXL/`.
21. A. Schürr, A. J. Winter, and A. Zündorf. *In [8]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
22. A. Singh and J. Billington. A formal service specification for IIOP based on ISO/IEC 14752. In B. Jacobs and A. Rensink (eds.), *Proc. Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pp. 111–126. Kluwer, Enschede, The Netherlands, 2002.
23. J. Sprinkle and G. Karsai. Defining a basis for metamodel driven model migration. In *Proceedings of 9th Annual IEEE Internation Conference and Workshop on the Engineering of Computer-Based Systems, Lund, Sweden*. 2002.
24. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In J. Padberg (ed.), *UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques*, vol. 44 (4) of *ENTCS*. 2001.
25. D. Varró. Automatic program generation for and by model transformation systems. In H.-J. Kreowski and P. Knirsch (eds.), *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 161–173. Grenoble, France, 2002.

26. D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 378–392. Springer-Verlag, Barcelona, Spain, 2002.
27. D. Varró, S. Gyapay, and A. Pataricza. Automatic transformation of UML models for system verification. In J. Whittle et al. (eds.), *WTUML'01: Workshop on Transformations in UML*, pp. 123–127. Genova, Italy, 2001.
28. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.
29. World Wide Web Consortium. *MathML 2.0.* URL {`http://www.w3c.org/Math`}.