Softw Syst Model (2003) 2: 187–210 / Digital Object Identifier (DOI) 10.1007/s10270-003-0028-8

VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML*

(The Mathematics of Metamodeling is Metamodeling Mathematics)

Dániel Varró, András Pataricza

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

Received: 18 February 2003/Accepted: 16 June 2003
Published online: 28 August 2003 – © Springer-Verlag 2003

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 multilevel 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

Recently, the main trends in software engineering have been dominated by the **Model Driven Architecture (MDA)** [26] vision of the Object Management Group (OMG). According to MDA, software development will be driven by a thorough modeling phase where first (i) a *platform independent model* (PIM) of the business logic is constructed from which (ii) *platform specific models* (PSMs) including details of the underlying software architecture are derived by *model transformations* followed by (iii) an automatic generation of the target application code.

The PIMs and PSMs are defined by means of the Unified Modeling Language (UML) [33], which has become the *de facto* standard visual object-oriented modeling language in systems engineering 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 to customers in addition to system engineers and programmers).

However, based upon academic and industrial experiences, recent surveys (such as [18]) have pinpointed several shortcomings of the language concerning, especially, 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 needed 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 aim at an *in-depth* evolution of UML into a core kernel language (UML Infrastructure 2.0), and an extensible family of distinct languages (UML Superstructure). According to these proposals, each UML sublanguage should have its own (individually defined) semantics, which fundamentally requires an appropriate and precise metamodeling technique.

* 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 the University of Paderborn (Germany), and it was funded by the National Science Foundation Grant (CCR-00-86096), the SEGRAVIS Research Network, and Hungarian Scientific Grants FKFP 0193/1999, OTKA T038027, and IKTA 00065/2000.

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 sublanguage for Schedulability, Performance and Time [24]. 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 a (rule-based) *operational 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* (or *denotational semantics*).

Concerning model transformation in a UML environment, the main stream of research is dominated by two basic approaches: (i) transformations specified in (extensions of) UML and the Object Constraint Language (OCL, [27]) [2, 3, 9], and (ii) transformations defined in UML and captured formally by graph transformations [15, 43]. Up to now, OCL-based approaches typically superseded graph transformation-based approaches when considering multilevel static metamodeling aspects (as the latter is traditionally restricted to a type-level and and instance level). However, when defining dynamic semantics of a modeling language, graph transformation has clear advantage over OCL due to its visual and operational (if-then-else like) nature – where, in fact, the formal technicalities of the underlying mathematics are hidden.

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 [7]) demonstrated that automated transformations of UML models into various semantic domains (including Petri nets, Kripke automata, 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 [11, 43] (a prototype automated model transformation system), we managed to overcome these problems by providing an *automated methodology for designing transformation* of UML models. After implementing more than 10 rather complex transformations in this methodology (including model transformations, for instance, for automated program generation [39], static consistency analysis [29] and model checking [19] for 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 transformations of UML sublanguages into executable platform/code or appropriate semantic domains (i.e., the abstract syntax of a UML model is mapped into such as Petri nets, finite automaton, etc.), the proper handling of formal mathematics is indispensable for developing automated and highly portable tools for analysis and code generation in the MDA environment.

Metamodeling mathematics. Meanwhile, recent standardization initiatives (such as PNML [1], GXL [34], GTXL [37], or MathML [45]) aim at developing XML based description formats for exchanging models of mathematical domains between different tools. Frequently (as e.g. in [37]), 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). On the other hand, as demonstrated in [12] (where dependability analysis of BPM-based e-business applications is carried out with dataflow networks as the mathematical background), a well-constructed metamodel could drastically reduce the time and workload related to the implementation of even complex mathematical analysis tools.

Our contribution. We first demonstrate (in Sect. 2) that although the overall goals of MOF metamodeling [25] (which is the existing industrial metamodeling standard)

are highly relevant for the specification and integration of modeling languages, the traditional metamodeling foundations and concepts of MOF (like the four-layer architecture itself) are inappropriate from many aspects.

As the main contribution of the paper (Sects. 3 and 4), we propose a visual but mathematically precise metamodeling framework (abbreviated in the following as VPM: Visual and Precise Metamodeling) based upon the structure of mathematical definitions (for the abstract syntax) and graph transformation (for dynamic operational semantics). Starting from a very concise (thus easy-to-implement) kernel language VPM builds up a hierarchy of models and modeling languages satisfying the rules of a refinement calculus that handles the most important features of the current (and upcoming) metamodeling standard but avoids the problems identified in Sect. 2

In addition, a static consistency analysis technique is introduced (in Sect. 5) to automatically detect (and partially correct) contradictions in the refinement hierarchy during the evolution of either a model or a modeling language.

Finally, we demonstrate (by running examples and the case study of Sect. 6) that even abstract mathematical models can be understood by engineers if they are presented and specified visually by means of metamodels and graph transformation.

2 Problems of MOF metamodeling

At first, we briefly identify (or revisit) some major problems of MOF metamodeling that hinder the use of MOF as the ultimate technique for specifying modeling languages of arbitrary domains in a hierarchical and reusable way. These problems are presented as small situations that cannot be appropriately captured by the MOF metamodeling standard. However, they either contradict with the goals of MOF (like e.g., "a core metamodeling language should use a minimal number of elementary concepts"), or they present problematic situations in modeling practice that are not even addressed by MOF.

Lack of package (metamodel) inheritance. Many existing UML profiles clearly demonstrate that the most general concepts (like events, actions, constraints, basic types, etc.) are redefined over and over again for many different profiles. This problem stems from the fact that MOF metamodels cannot be arranged in a refinement hierarchy, thus domain experts responsible for the creation of a specific profile cannot build upon a reusable abstract metamodel library. Such a metamodel hierarchy is in analogy with meta-level design patterns that would encapsulate and reuse best engineering (and mathematical) practice in language design. In fact, a proper metamodeling technique would simultaneously handle both meta-level and model-level design patterns.

The key notion of such a hierarchy is captured as package inheritance in the MML approach [9], which extends the inheritance mechanism of classes to entire metamodels (encapsulated as a package). However, we demonstrate in Sect. 3.1 that the underlying concepts come from far beyond, namely, from the structure of mathematical definitions where a new notion is defined on the basis of an existing one (with the Zermelo–Frankel set theory on the top to provide meta-circularity in the engineering sense). For instance, each mathematical textbook on graph theory first introduces the notions of a graph, and then different subdomains (like bipartite graphs, planar graphs, etc.) are derived by restrictions (e.g., " a bipartite graph is a graph with..." as expressed in Fig. 1).

Fortunately, the latest proposal for the upcoming UML 2.0 standard [38] seems to provide some advanced concepts (such as redefine, import, package merge) to capture such a reusable metamodel hierarchy.

Lack of association inheritance. As MOF (and UML) evolved from traditional object-oriented programming languages only the inheritance of classes is allowed. However, the lack of an inheritance concept for associations hinders the development of reusable metamodel template libraries, since the user (domain expert) has to write additional well-formedness constraints (e.g., in OCL) to express that associations in the library should be restricted to lead between the derived classes of the new metamodel. Unfortunately, this is very error prone when compared to a proper inheritance mechanism for associations, since if such well-formedness constraints are omitted, one might

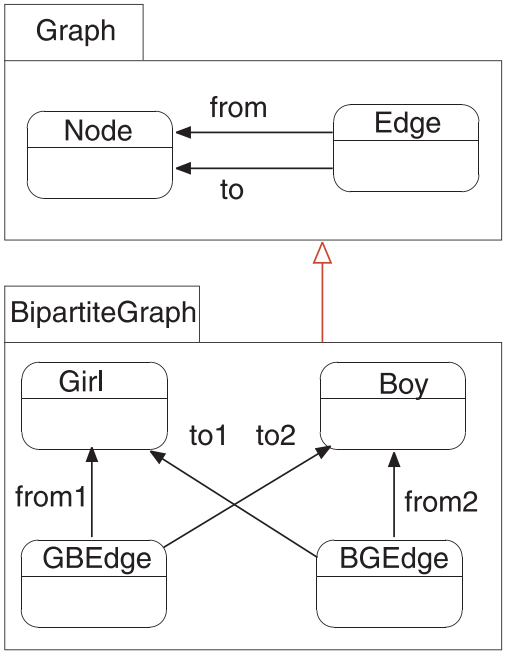


Fig. 1. Metamodel (package) inheritance: a bipartite graph is a graph

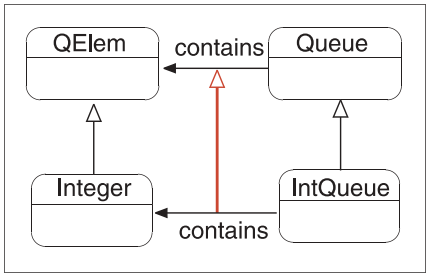


Fig. 2. Association inheritance

not be able to detect that an instance model does not conform to its metamodel.

The metamodel of queues in Fig. 2 specifies that a *Queue* object may contain elements of class *QElem*. Now, if queues are aimed to be reused to contain only integers, then one can state in MOF that an integer queue *IntQueue* is queue, and the class *Integer* is a subclass of *QElem*, but without association inheritance, the fact that an integer queue may only contain integers as elements can only be expressed by explicit OCL constraints.

Structural redundancies in MOF. Even though MOF should provide a minimal set of kernel constructs that is required to specify modeling languages in a hierarchical way, both the current MOF and the upcoming UML 2.0 core is redundant concerning containment and attributes.

As demonstrated in Fig. 3, one can express the mathematical fact that *name* is a function that maps *States* to *Strings* as (i) introducing *name* as an attribute of class *State*, or (ii) using an association with exactly one (or at most one in case of partial functions) multiplicity at the navigable *String* end.

Moreover, expressing containment for classes is also redundant as we can alternatively use package containment and aggregations (for specifying that, for instance, a *Statemachine* contains *States*). In many cases (concerning reusable metamodels), it is extremely hard to judge whether a package or a class (or probably both) is required for a certain concept.

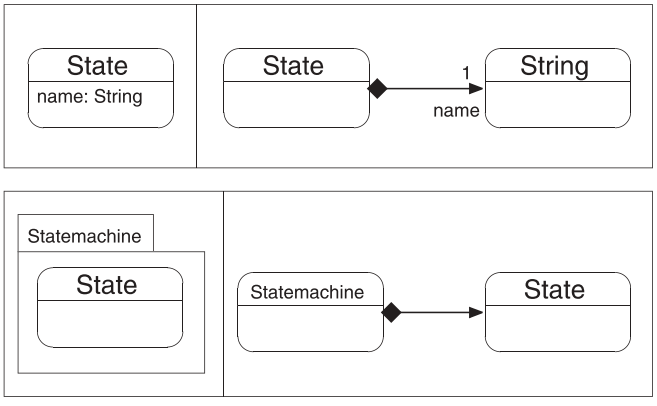


Fig. 3. Structural redundancy in MOF

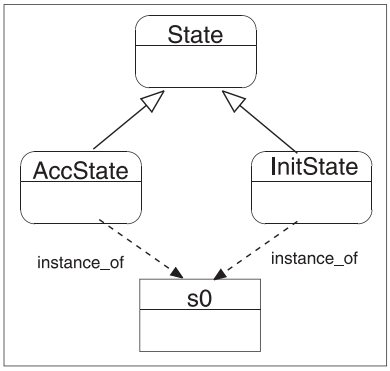


Fig. 4. Multiple instantiation

Lack of multiple instantiation. MOF (and UML) follows traditional type theoretic foundations of object-oriented programming languages where even if multiple inheritance is allowed, objects are only permited to have a single class as their direct type.

However, Fig. 4 depicts a simple example to highlight the essence of the problem. Let us suppose that accepting states *AccState* and initial states *InitState* are two subclasses of *State* in the language of finite automata. Then a state instance which is simultaneously accepting and initial can only be created if an object is allowed to have multiple types. In Sect. 3.2, we demonstrate that multiple instantiation can be handled identically to multiple inheritance.

Unfortunately, the same problem appears throughout in the UML environment as entire extension mechanism of UML (based on profiles and stereotypes) is chaotic (see [6] for an overview of the diversity of proposals that formally capture stereotyping) due to the lack of multiple instantiation. In principle, a UML profile (like SPEM [22], EDOC [23], or GRM from the UML Profile for Schedulability and Time [24]) is a modeling language (metamodel) designed for certain domain created by experts in order to model the target application from additional aspects. However, *such a metamodel itself is totally independent of UML*, in other terms, UML is just one language that a domain profile could be tailored to but could possible be reused in other modeling languages. For instance, resource usage can be modeled by the GRM formalism [24] not only for a software application (embedding it in UML) but also for control applications (e.g. combining with Matlab/Simulink).

A proper metamodel-based handling of the problem is to instantiate constructs in a user model from multiple modeling languages. Note that allowing multiple in the modeling phase is very consistent with the well-known "separation of concerns" principle (i.e., one can safely refer to the same model element from multiple aspects by using multiple modeling languages). For this reason, it would be good *modeling* practice even if it is not directly supported in an *implementation* phase by existing programming languages.

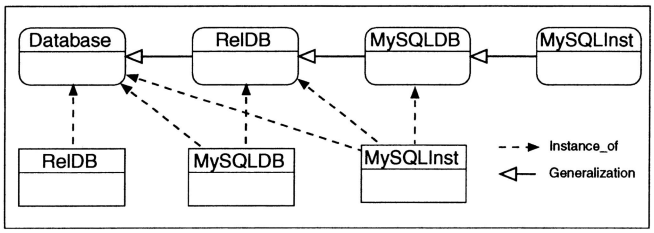


Fig. 5. Problems with metalevels

Problems with metalevels. As discussed previously in [5], there are fundamental problems with the traditional four-layer MOF architecture. In many cases, same *concepts are replicated* both on meta-level and model-level (or other two adjacent layers) as (meta)classes can only be instantiated one level down (up?) in the hierarchy (*shallow instantiation*).

In the paper (see also Sect. 3.2), we argue that the problem is directly caused by the fact that the metamodel derivation process is quantitized into four discrete levels, moreover, the borders of such metalevels are fixed. As a result, one has to artificially distinguish between classes and objects (meta-level and model-level instances) of the same real world entity, which doubles the size of the model space.

The problem relies in the fact that during different modeling phases, the same concepts can be regarded both classes *and* instances as well. For instance, when modeling databases in Fig. 5, we can simultaneously say (typically, on different level of abstraction) that a relational database *RelDB* is both a subclass *and* an instance of databases (see the generalization and instance-of relations leading to *Database*). According to the traditional MOF concepts, we need to create a separate class and object for the same real world concept.

Metamodeling vs. software engineering. As a conclusion, many of these problems are probably *not crucial* in a general purpose *modeling language* (like UML) *for designing software applications* (as they might be too abstract for an average systems engineer). However, they demonstrate *major weaknesses of a metamodeling kernel language* used for designing other modeling languages by domain experts. Unfortunately, many of the previous problems are left unhandled even in the latest proposal for the upcoming UML 2.0 core language [38].

As UML has been used in practice for many years now with a wide range of existing applications, resolving such problems with minimal changes in the standard is very difficult. In fact, there are two rather complementary conceptual solutions: (i) one is to separate the techniques for designing applications and modeling languages, which may keep UML relatively unaltered but contradicts with the MDA vision (saying that models and modeling languages are designed within the same modeling approach, i.e., UML), (ii) the other is to adapt the changes in UML as well, which would result in a major redefinition of (at least) its infrastructure.

Although we propose rather radical changes to the underlying metamodeling concepts, an intuitive and simplified UML/MOF notation is used in the paper to emphasize that changes in the depth (semantics of metamodeling) do not necessarily involve changes on the surface (syntax of metamodels), and if so, these changes are mainly simplifications of the existing MOF standard. Moreover, as VPM is a multilevel approach the original MOF metametamodel can be integrated into our framework as any other modeling languages thus the conformance with the current version of the standard can also be maintained.

3 Structural refinement of metamodels

Below we define a structural refinement calculus on set theoretical basis (i.e., refinement of sets, relations, functions and tuples) for major MOF (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 the previous metamodeling problems we introduce *dynamic (or fluid) metalevels* where the type–instance relationship is derived between models instead of explicitly predefining it by (meta)levels. Our approach has the major advantage that the type–instance relations can be reconfigured dynamically throughout the evolution of models, thus transformations on (traditional) model and metamodel "levels" can be handled uniformly.

3.1 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 [36]). From an UML point of view, transforming UML models to Petri nets provide dependability [8] and performance analysis [17] for the system model in early stages of design.

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

Definition 1. *A simple 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**).*

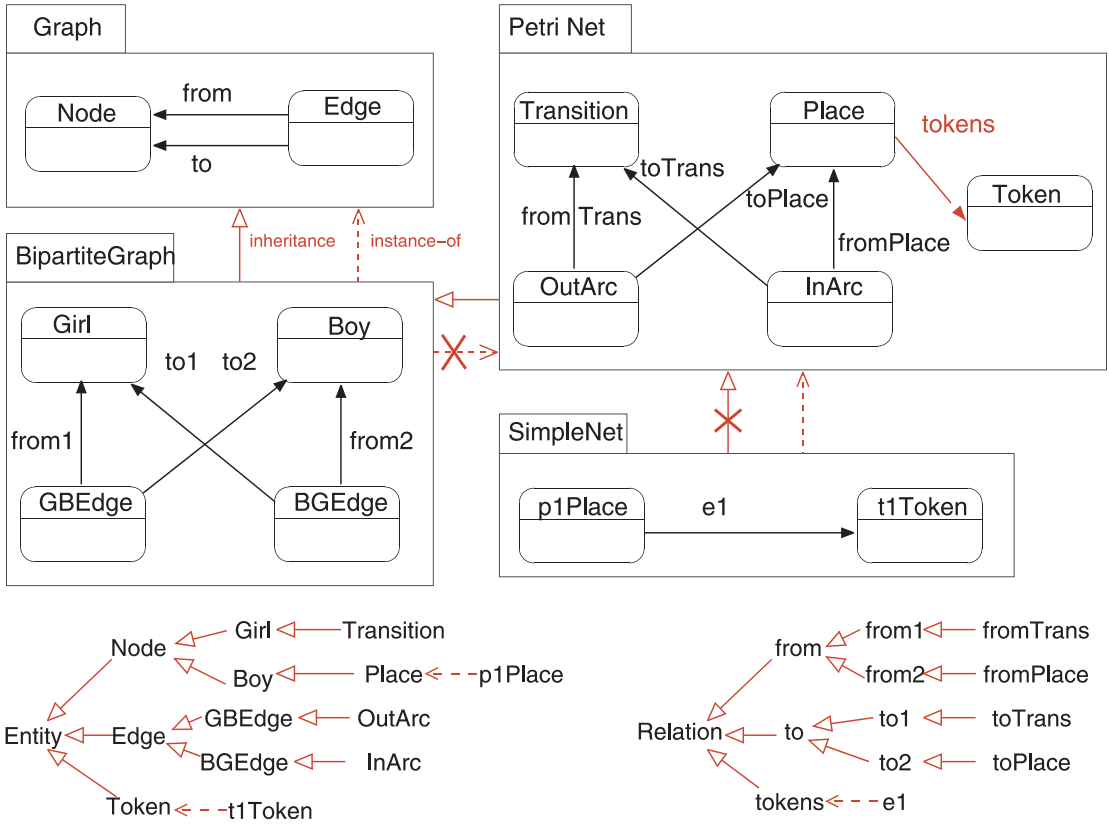


Fig. 6. Defining the structure of Petri Nets

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 *Petri Net* package in the upper right corner of Fig. 6) 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. 6) 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. 6).

First, we intend to inter-relate these metamodels in such a way to be able to express that, for instance, (i) the class *Node* is a supertype of class *Boy*, and (ii) the association *fromPlace* is inherited (indirectly) from the association *from*. As a result of such elementary inheritance relations, we would also like to state that the metamodel of bipartite graphs 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 *SimpleNet* package in the lower right corner of Fig. 6, a Petri net model 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 (see Sect. 3.2), and (ii) the dynamic operational semantics of models can also be inherited and reused with an appropriate model refinement calculus (Sect. 4) in addition to the static parts of the models.

3.2 Formal semantics of static model refinement

Our VPM metamodeling framework uses a minimal subset of MOF constructs (i.e., classes, associations, attributes, and packages) with precisely defined semantics, which has a direct analogy with the basic notions of mathematics, i.e., sets, relations, functions, and tuples (tuples

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

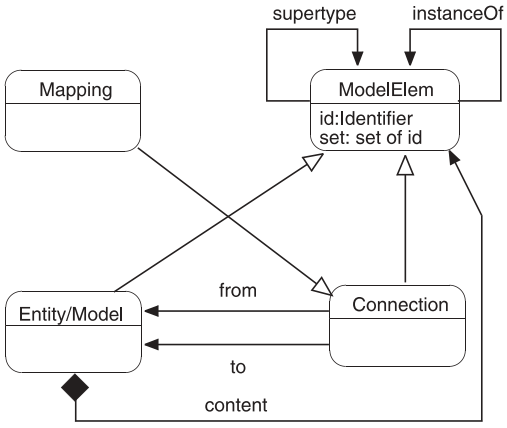


Fig. 7. The MOF metamodel of our approach

are constituted in turn from sets, relations and other tuples).

Modeling concepts. 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**, or a **mapping** (see the MOF metamodel of our approach in Fig. 7). **A unique identifier** (accessed by a *.id* postfix in the sequel) *and* **a set** including the *identifier of the model element* and the *identifiers of all the (intended) refinements of the element* (accessed by a *.set* postfix) are related to each of this constructs.[2] For mathematical reasons, the set associated to a model element should also contain the identifier of the element (to be able to detect circularities in typing later on).

- An **entity** E is a set (called as **basic entity** in this case) or a tuple (denoted as **compound entity** or **model**) consisting of sets, relations, functions and tuples (a collection of entities, connections, and mappings, respectively). Entities will be represented visually either by UML classes or UML packages while the notion of containment will be captured by graphical containment (e.g., classes inside a package) or aggregations (leading from entities to both entities, connections and mappings) depending on the context to provide the better match with the conventional notation.
- A **connection** R between two entities is a binary relation between the associated sets or tuples. Connections are depicted as (directed) UML 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 with an attribute type corresponding to its range. In a strict mathematical sense, either connection or mapping would be sufficient for a truly minimal metamodeling kernel; however, we introduced both to keep our concepts close simultaneously to UML/MOF and mathematical concepts as well.

A significant change in contrast to [42] is the merging of previously distinct notions of entities and models into a single entity construct, which idea stems from the fact that a one-dimensional tuple (consisting of only a single set) can be regarded as a set, thus a certain redundancy is eliminated from the underlying mathematical framework of our approach. In an object-oriented term, a uniform class concept is used for both classes and packages (models).

A refinement calculus for inheritance and instantiation. The static semantics of our metamodeling framework is based upon a unifying refinement calculus, which captures the notion of inheritance and type-instance relationship (depicted by UML generalization and instance-of relations, respectively) between arbitrary metalevels *without actually defining the notion of metalevels*.

Definition 2 (Comparison of elements). *A model element P (i.e., either entity, connection or mapping) is less than (or equal to) a model element Q (denoted as $P \leq Q$) iff $P.set \subseteq Q.set \wedge P.id \in Q.set$.*

Thus $P \leq Q$ holds if the related set of P is a subset of the corresponding set of Q and the identifier of P is contained by the set of Q, which defines an ordering relation combining the subset and set containment relations.

For the notational convention of Definition 3, let (i) $E^{(n)}$ denote an entity consisting of exactly n subcomponents (in case of $n = 1$, the entity is regarded to be *basic*, otherwise *compound*) where $E[i]$ accesses the ith component (argument) of entity E; (ii) $R(A, B)$ refers to a connection R between entities A and B; while (iii) $F(A, B)$ denotes a mapping with the domain of entity A and range of entity B.

Our general refinement relation between model elements Sub and $Super$ will be denoted as $Sub \sqsubseteq Super$[3] (consistently with the partial order imposed by the refinement graph later in Sect. 5), which means that Sub is a refinement of $Super$ (for the use of our terminology, see Table 1). Refinement is unified relation that is either an inheritance $\Rightarrow$ or an instantiation $\mapsto$ (or both).

Definition 3 (Refinement calculus). *The refinement ($\sqsubseteq$) rules of our metamodeling framework (that simultaneously handle inheritance $\Rightarrow$ and instantiation $\mapsto$) are as follows.*

[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.

[3] Note the difference between the $\subseteq$ symbol used as the traditional subset relation and the $\sqsubseteq$ symbol that denotes the refinement of VPM model elements.

Table 1. Notation guide for refinement, inheritance and instantiation

	sub is ... *super*	*super* is ... *sub*
$\sqsubseteq$: refinement = inheritance + instantiation	refinement of	abstraction of
$\Rightarrow$: inheritance (subtype)	inherited from (subtype of)	generalization of (supertype of)
$\mapsto$: instantiation (type-instance relation)	instance of	type of

1. **Basic entity refinement**:
   $E^{(n)}_{sub} \sqsubseteq E^{(1)}_{super} \stackrel{\text{def}}{=} E^{(n)}_{sub} \leq E^{(1)}_{super}$, thus if E_{super} is a simple entity (one-dimensional tuple consisting only of a set) then refinement is defined as the "less-than" relation. Informally, E_{super} is either a generalization or a type of E_{sub} (which is either a class or a package) if using a MOF metamodeling analogy.
2. **Connection refinement**:
   $R_{sub}(A_{sub}, B_{sub}) \sqsubseteq R_{super}(A_{super}, B_{super}) \stackrel{\text{def}}{=} R_{sub} \leq R_{super} \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$ (where all A_i and B_i are entities). Connection inheritance expresses the fact that MOF associations can also be refined during the evolution of metamodels in addition to the refinement of classes.
3. **Mapping refinement**:
   $F_{sub}(A_{sub}) : B_{sub} \sqsubseteq F_{super}(A_{super}) : B_{super} \stackrel{\text{def}}{=} F_{sub} \leq F_{super} \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$ From a practical point of view, the refinement of MOF attributes is also handled in our metamodeling framework (similarly to classes and associations).
4. **Mapping is connection**:
   $F(A_{sub}) : B_{sub} \sqsubseteq R(A_{super}, B_{super}) \stackrel{\text{def}}{=} F \leq R \wedge A_{sub} \sqsubseteq A_{super} \wedge B_{sub} \sqsubseteq B_{super}$, i.e., functions can be interpreted as special relations. In practical uses of this axiom, traditional cardinality restrictions in MOF metamodels can be strengthened such as the cardinality of a role can be changed from "arbitrary number" (*0..**) to "at most one" (*0..1*).
5. **Compound (model) refinement**:
   the refinement of compound entities (or models/packages) is explicitly split into the inheritance and instantiation case, thus $E^{(n)}_{sub} \sqsubseteq E^{(k)}_{super} \stackrel{\text{def}}{=} E^{(n)}_{sub} \Rightarrow E^{(k)}_{super} \vee E^{(n)}_{sub} \mapsto E^{(k)}_{super}$

   (a) **Compound entity (model) inheritance**
   $E^{(n)}_{sub} \Rightarrow E^{(k)}_{super} \stackrel{\text{def}}{=} E_{sub} \leq E_{super} \wedge \forall i \exists j : E_{sub}[j] \Rightarrow E_{super}[i]$. Informally, there exists a subtype relation for each argument of E_{super} in a corresponding argument of E_{sub}. In MOF terms, each class in the super package is refined into an appropriate class of the subpackage. However, this latter one may contain additional classes not having origins in the super package.

   (b) **Compound entity (model) instantiation**
   $E^{(n)}_{sub} \mapsto E^{(k)}_{super} \stackrel{\text{def}}{=} E_{sub} \leq E_{super} \wedge \forall i \exists j : E_{sub}[i] \mapsto E_{super}[j]$. Informally, there exists a *type* element for each component of E_{sub} in a corresponding component of E_{super}. In MOF terms, each object in the instance model has a proper class in the metamodel. However, the metamodel one may contain additional classes without objects in the instance model.

The most crucial consequence of these definitions is that the handling of refinement (inheritance and instance-of) relations is identical for basic entities, connections and mappings (while there is a certain orthogonality for compound entities/models). As a result, *two model elements can simultaneously be in subtype and instance-of relations*, which is a major difference with the MOF standard.

In order to obtain a mathematically complete definition of our refinement calculus (that encapsulates the metametamodel of Fig. 7 within our framework), a top section of the inheritance and containment hierarchy is introduced as follows.

Definition 4. *The model space of our framework will always consist of (at least) the following elements.*

- *The abstract model element* Universe *or* Top *is greater than all the other model elements (entities, connections and mappings), thus being the root of both the inheritance and the containment hierarchy.*
- *The entity* Entity *is contained by (and refined from)* Universe.
- *The connection* Connection *is leading from and to* Entity.
- *The mapping* Mapping *is leading from and to* Entity.

Thereafter, any well-formed model space has to fulfill the following axioms.

Property 1 (Inheritance and instantiation is partial order). Each element in the model space (except for *Universe*) has at least one supertype, and both refinement relations (inheritance and instantiation) are *reflexive*, *transitive* and *anti-symmetric*.

Informally, multiple inheritance and instantiation are allowed but circularities are therefore forbidden in the type hierarchy (to be precise, treated as equality). Note that the definition formally permits that an element is inherited from (alternatively, instantiated from) itself, which only eases the mathematical treatment of our framework without foregoing consequences on the intuitive meaning.

Property 2 (All model elements are contained). Each element in the model space (except for *Universe*) is contained by at least one element, moreover, the containment relation is *transitive*.

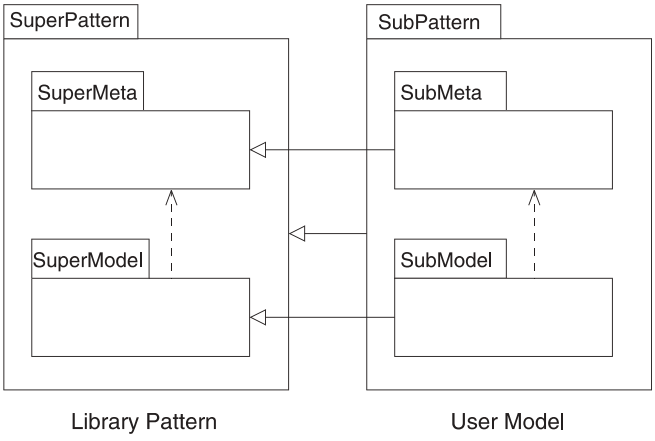


Fig. 8. Pattern refinement

As a consequence, multiple and circular containment are thus allowed by this axiom, and each element should be reachable from the top element by navigating containment relations.

Pattern refinement. We introduce the notion of pattern refinement as a special case of entity refinement, which provides a means to formally capture the use of design patterns[4] (i.e., how an abstract pattern can be embedded in a concrete user model) by entity refinement. Moreover, pattern refinement will also form the bases of rule refinement later in Sect. 4.3

The overall idea (and typical use) of pattern refinement is depicted in Fig. 8 where *SubPattern* is intended to be a refinement ($\sqsubseteq$) of *SuperPattern*.

We suppose that the abstract pattern *SuperPattern* stored in a pattern library is a compound entity containing a "metamodel" entity *SuperMeta* specifying type information and a model entity *SuperModel* (which is an instance of *SuperMeta*) describing the designated use of the pattern. Thereafter, in a concrete user model (*SubModel* in package *SubPattern*) aiming to apply the pattern properly in a application domain defined by the metamodel *SubMeta*, one has to establish the inheritance relation between *SuperPattern* and *SubPattern* by showing that *SuperMeta* is a generalization of *SubMeta* (i.e., the application domain is a proper refinement of the pattern metamodel) and *SuperModel* is a generalization of *SubModel* (i.e., the user model contains at least the elements required by the library pattern *SuperModel*).

As a summary, we can formally define patterns and pattern refinement as follows.

Definition 5 (Pattern). *A* ***pattern*** P *is a compound entity consisting of entities* $Meta$ *and* $Model$ *where* $Model \mapsto Meta$.

[4] Essentially, the use of architectural styles can be handled similarly to design patterns.

Definition 6 (Pattern refinement). *A pattern* P_{sub} *is a* ***pattern refinement*** *of* P_{super}*, if* $P_{sub} \sqsubseteq P_{super}$.

The four-layer MOF architecture in VPM. A main advantage of our approach (in contrast to e.g., [4] or the MOF standard itself) 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 all the problems of Sect. 2. 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.

Furthermore, our metamodeling framework clearly demonstrates that the fixed number of metalevels introduced by the MOF standard is artificial and mathematically unsound: the four layers are finite restrictions of a general refinement relation, which is transitive both in case of inheritance and instantiation.

On the other hand, as VPM is more general than MOF, the original four layer MOF architecture can be embedded into VPM by introducing the following constraints (in fact, they can be captured in a constraint language like OCL as static well-formedness rules).

1. The metametamodel defined by MOF can be introduced as a compound entity refined from our top-level concepts.
2. The metamodels of modeling languages (like the UML metamodel, Petri Net metamodel etc.) are entity instances of the MOF metametamodel entity.
3. User models can be instances of the metamodel entities.
4. The object level instances have in turn related entities to corresponding user models.
5. Within the same "metalevel", arbitrarily long chains of inheritance relations are allowed.

3.3 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. 6 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(n entity) subtype and instance of* Graph.

Proof. The proof consist of two steps.

1. **Proof of refinement:** for each element in the *Graph* model there exists 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 exists an instance-of relation in *Graph*. *Girl* and *Boy* are instantiations 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. 6. Note that *Petri Net* is *not an instance of* *Bipartite Graph* (as *Token* is a new element), and *SimpleNet* is *not inherited from* *Petri Net* (since, for instance, there are no transitions).

4 Dynamic refinement in operational semantic rules

Now we 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* [43] 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 merely provide a pattern-based manipulation of models driven by a statechart diagram (specified, in fact, by a UML profile in our VIATRA tool), thus a 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 (as an example) 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 in analogy with the well-known concepts of dynamic binding and operator overloading in traditional object-oriented languages.

4.1 An introduction to model transition systems

Graph transformation (see [32] 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. 9 for demonstration) is a triple $Rule = (Lhs, Rhs, Neg)$, 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). As all graphs in a graph transformation rule have to be well-typed, they can be regarded as *patterns* (see Definition 5).

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. **Match.** *finding a matching* of the Lhs pattern in model M;
2. **Check.** *checking the negative application conditions* Neg which prohibits the presence of certain model elements;
3. **Delete.** *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. **Glue.** *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 define 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** [43], where the permitted 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) in parallel for all matches as in case of **forall** mode; (ii) on a (nondeterministically selected) single matching as in case of **try** mode; or (iii) as long as applicable (in **loop** mode).

4.2 Model transition system semantics of Petri nets

In order to demonstrate the expressiveness of our semantic basis (and the technicalities of graph transformation), we provide a model transition system semantics for Petri nets (in Fig. 9) based on the following definition.

Definition 7 (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

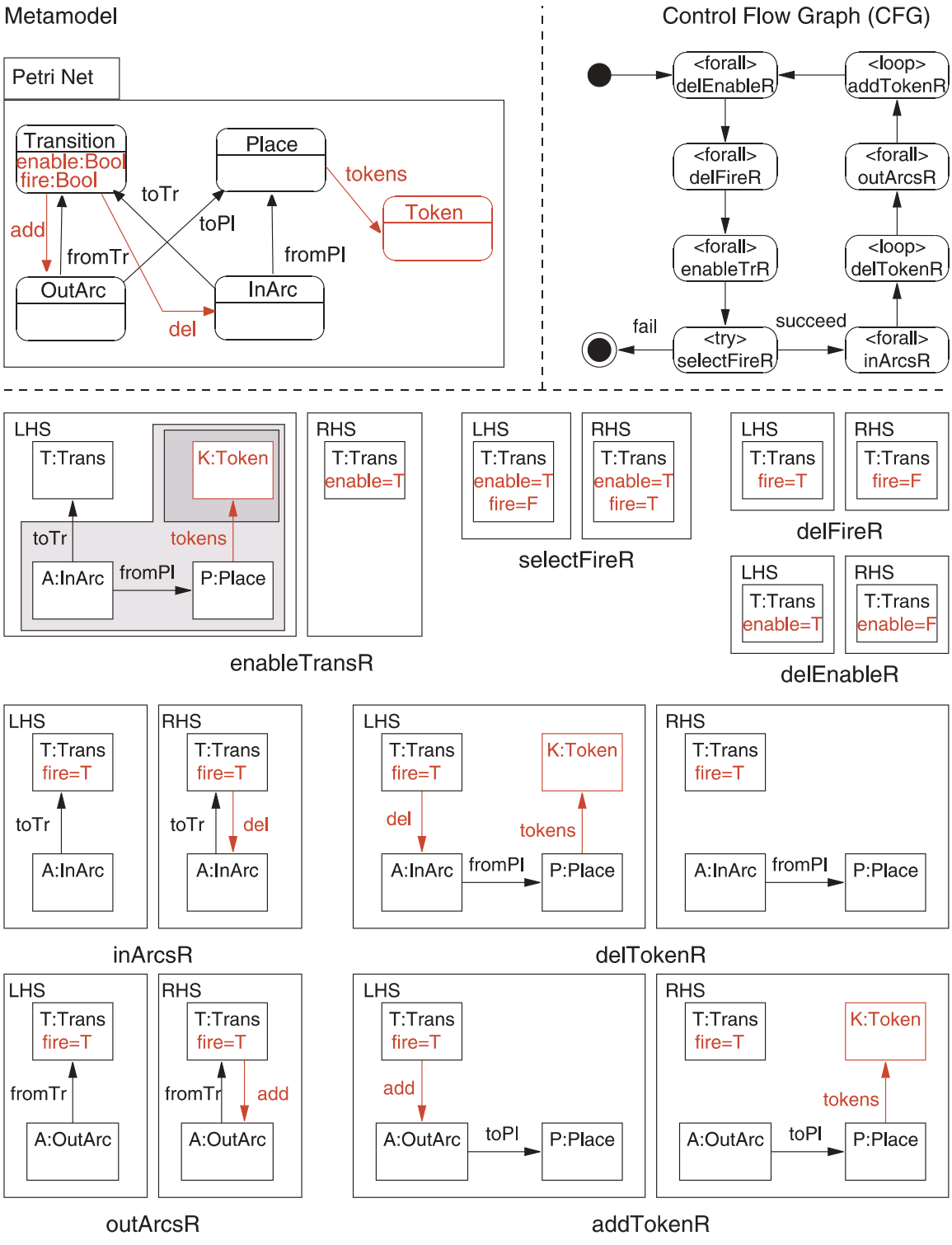


Fig. 9. Model transition semantics of Petri nets

parts. Then the informal interpretation of the rules (given in the order of their application) is as follows. In order to improve clarity, the types of each model element are depicted textually instead of the original graphical representation by instance-of relations. Thus, for instance, *T:Trans* can be interpreted as *T* is a direct instantiation of *Trans*.

1. First, 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 arcs *A* linked to a place *P*, this place must contain at least one token *K* (note the double negation in the negative conditions).

3. A single transition is selected non-deterministically by executing *selectFireR* in *try* mode. If no transitions are enabled then the execution 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.

4.3 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 (also 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 adaptable for this specific domain without further modifications.

Moreover, semantic operations are frequently needed to be organized in a hierarchy (and 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. Thus a domain engineer should be able to derive the "text copying" operation from the abstract "copy a thing to a certain place" operations. In other terms, in analogy with traditional structural ontologies, dynamic behavior can also be classified into a hierarchy and reused in specific application domains.

To capture such semantic abstractions, we define *rule refinement* as a precise extension of metamodeling for dynamic aspects of a domain as follows, which is conceptually similar to the use of late bindings and method overriding in object-oriented programming languages.

Based upon the definition of pattern refinement (see Definition 6), the refinement relation of typed rules is defined as follows:

Definition 8 (Rule refinement). *A rule* $r_{sub} = (Lhs_{sub}, Rhs_{sub}, Neg_{sub})$ *is a* ***refinement*** *of rule* $r_{super} = (Lhs_{super}, Rhs_{super}, Neg_{super})$*, denoted as* $r_{sub} \sqsubseteq r_{super}$*, if*

1. $Lhs_{sub} \sqsubseteq Lhs_{super}$: the positive preconditions of r_{super} are not stronger (more general) than of r_{sub};
2. $Neg_{sub} \sqsubseteq Neg_{super}$: the negative preconditions of r_{super} are not stronger than of r_{sub};
3. $Lhs_{sub} \cap Rhs_{sub} \sqsubseteq Lhs_{super} \cap Rhs_{super}$: the preserved elements of r_{super} are not stronger than of r_{sub}
4. $Lhs_{sub} \setminus Rhs_{sub} \sqsubseteq Lhs_{super} \setminus Rhs_{super}$: thus r_{sub} removes at least the elements that are deleted by the application of r_{super};
5. $Rhs_{sub} \setminus Lhs_{sub} \sqsubseteq Rhs_{super} \setminus Lhs_{super}$: thus r_{sub} adds at least the elements that are added by the application of r_{super}.

As a direct consequence of this definition, the following proposition can be established.

Proposition 2. *If* $r_{sub} \sqsubseteq r_{super}$*, (i.e.,* r_{sub} *is a structural rule refinement of* r_{super}*), then* r_{super} *can be applied whenever* r_{sub} *is applicable (which means a certain refinement of dynamic behavior).*

Examples on rule refinement. The concepts of rule refinement are demonstrated on a brief example (see Fig. 10). Let us suppose that a garbage collector removes a *Node* from the model space (by applying rule *delNodeR*) if the reference counter of the node (which collects the number of edges leading into the node) has been decremented to 0 (denoted by the attribute condition *ref=0*). Thus a transformation sequence for garbage collection may only remove isolated nodes from the model space.

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 checking 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 3. *Rule* delPlaceR *is a refinement of rule* delNodeR*. Therefore, in typical applications (visual model editors, etc.)* delPlaceR *takes precedence of* delNodeR*.*

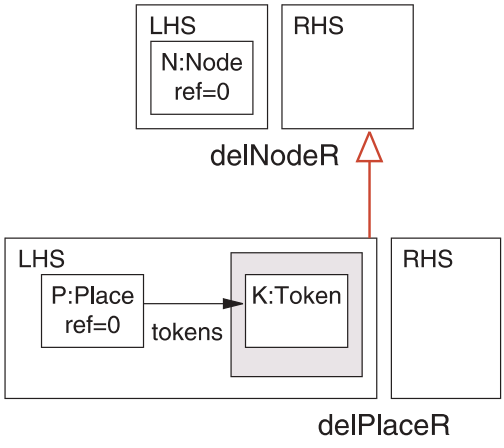


Fig. 10. Rule refinement

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

1. $Lhs_{delPlaceR} \sqsubseteq Lhs_{delNodeR}$ since *Place* (the type of *P*) is a refinement of *Node* (the type of *N*).
2. $Neg_{delPlaceR} \sqsubseteq Neg_{delNodeR}$ since rule *delNodeR* has no negative conditions.
3. Conditions 3–5 trivially hold due to the empty right-hand sides of rules.

For a more complex example, let us consider that we want to derive from the general Petri net metamodel the specific metamodel of 1-bounded or 1-safe Petri nets where each place may only contain at most one token at a time. Therefore, some slight changes are needed to be introduced in the Petri net semantics to capture that a transition is only enabled if all incoming places contain a token but none of the outgoing places (that are not incoming places as well) contains a token.[5]

Our goal is to maintain as much as possible from the original Petri net formalization in Fig. 9, and introduce a new rule (Fig. 11) by rule inheritance that modifies the semantics in the right way. As a result, the dynamic operational semantics of a language will also be reused when defining a more refined language (in addition to reusing its structural definition).

– As for structural changes in the metamodel of 1-safe Petri nets, the *tokens* connection is refined into a *tokens* mapping (as indicated by the highlighted areas).

[5] From a mathematical point of view, this subclass of Petri nets can be simulated by the original semantics as well with a structural modification that introduces a complementary place for each existing one.

– In the dynamic parts, the only new rule is *safeEnableR*, which is a proper refinement of the former version of the rule *enableTransR*, since a new negative condition has been introduced that prescribes that no outgoing places (see place *P1*) are permitted to contain a token *M* prior to enabling the transition *T*, except for those that are also serve as incoming places for the same transition (connected by *InArcs* as well). One can easily check that the rule refinement relation holds between rules *enableTransR* and *safeEnableR* by the following argument.
  – $Lhs_{safeEnableR} \sqsubseteq Lhs_{enableTransR}$, since they are identical (and the refinement of *tokens* from a connection to a mapping is valid).
  – $Neg_{safeEnableR} \sqsubseteq Neg_{enableTransR}$, since an additional negative condition has been introduced while all existing conditions are left unaltered.
  – Conditions 3 and 4 trivially hold as the right-hand sides of rules are identical.

As a result, we enabled the reuse of dynamic operational semantics of a modeling language by providing a precise means of refining graph transformation rules in addition to the conventional reuse of structural elements.

Transformations between modeling languages. Graph transformation rules frequently serve as a visual but mathematically precise way to capture model transformations between modeling languages [15, 43]. In a UML environment, such transformations typically include the mapping of UML models into semantic domains or into executable target code.

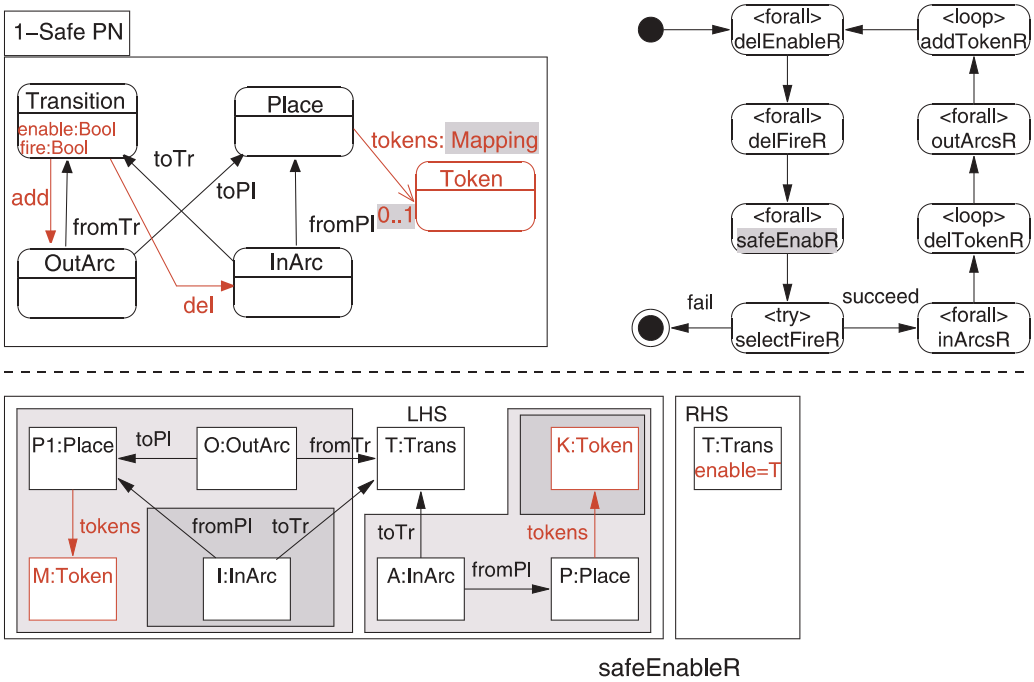


Fig. 11. 1-safe Petri nets: modifying semantics by rule refinement

Although a detailed discussion of this model transformation problem is out of the scope for the current paper (see [7, 11, 43] for further details of our approach), it is worth noting that the VPM formalism provides a direct support for such transformations.

1. We first construct the VPM metamodel of the source language (which is typically UML itself) and target language (i.e., the metamodel of the mathematical paradigm such as Petri nets, dataflow nets; or the metamodel of a programming language).
2. Then we design a reference metamodel that captures the interconnections of source and target model elements. Since entity containment is not strict in VPM (i.e., a model element can be contained by multiple entities), one can easily reuse the source and target metamodels for this purpose.
3. Later, we construct a model transition system to formalize the model transformation problem. In VPM terms, we provide an *operational semantics* for this *reference* metamodel. These graph transformation rules typically preserve all the constructs in the source language and generate new elements only in the target language.
4. Graph transformation rules are directly executable therefore the concrete model transformation process can largely be automated (using, for instance, the VIATRA tool [11]).

5 Static consistency analysis of metamodels

Although, in the Sect. 3, the concepts of models and metamodels have been formalized precisely, in the sequel, we introduce an equivalent representation (called *refinement graphs*) to visualize and automatically detect flaws in the refinement hierarchy. Refinement graphs eliminate the distinction between entities, connections and mappings, however, all type information is preserved. After that, we show (i) how one can judge whether a certain model is (in a consistent way) more abstract or more refined than another one, and (ii) how certain inconsistencies in merging models (packages) can be corrected automatically based directly on the abstract representation.

The practical feasibility of our approach is demonstrated on formalizing advanced concepts of metamodeling including structural extensions, type restrictions and recent concepts from the upcoming UML 2.0 standard [38] (like import, redefine and package merge constructs).

5.1 Refinement graphs

Definition 9 (Refinement graph). *The* ***refinement graph*** $RG = (Nodes, Edges)$ *of a given model space is a directed graph defined as follows.*

- *A* ***refinement node*** $n \in Nodes$ *is either an entity, a connection, or a mapping.*
- *A* ***refinement edge*** $e \in Edges$ *(that can be interpreted informally as* ***implication****) leads from node* n_1 *to node* n_2 *(denoted as* $n_1 \stackrel{e}{\longrightarrow} n_2$*, or simply* $n_1 \longrightarrow n_2$*) when*
  1. ***Inheritance:*** *for the corresponding model elements of nodes* n_1 *and* n_2*, (the model element of)* n_1 *is (directly or indirectly) inherited from (the model element related to)* n_2*;*
  2. ***Instantiation:*** *for the corresponding model elements of nodes* n_1 *and* n_2*,* n_1 *is (directly or indirectly) an instance of* n_2*;*
  3. ***Source of Connection/Mapping:*** n_1 *is related to a connection (mapping) leading from the entity itself associated to* n_2 *or one of its subentities;*
  4. ***Target of Connection/Mapping:*** n_1 *is related to a connection (mapping) leading to the entity itself associated to* n_2 *or one of its subentities;*
- ***Extensions****: For the mathematical treatment, let* $n_\perp$ *be a node having only outgoing edges that is linked to all the other nodes, and let* $n_\top$ *be a node (corresponding to* Universe*) having only incoming edges leading from all other nodes.*

Informally, an entity node has an outgoing edge to all its "super" entity nodes (by merging the concepts of inheritance and instance of relations), while a connection (mapping) node has an outgoing edge to all its "super" connections (mappings) edges, plus all the "super"entities of its source and target entity node.

Example 1. In Fig. 12, the refinement graph of simple model space containing the concepts of *State* and *Transition* of a finite automaton can be observed. For the sake of clarity, the model space is incomplete (not well-formed) in the sense that containment relations are not depicted.
The figure in the middle explicitly depicts all the edges of the refinement graph (except for self loops, and extension edges leading from $n_\perp$ and to $n_\top$). The figure in the right depicts (which will be our standard notation for the rest of the paper to improve clarity) only the *direct* refinement edges. The entire set of edges can be calculated by the transitive (and reflexive) closure of the explicitly depicted edges.
Moreover, the edges with a white arrowhead (like UML generalizations) will refer to inheritance and instantiation relations in the model (instantiation edges are labeled with *inst*), while edges with ordinary arrowhead (like navigable associations in UML) denote source and target restrictions (labeled with *src* and *trg*, respectively) for connections and mappings.
According to a final notational convention, nodes derived from entities appear in black, nodes related to connections in white, while nodes of mappings have a striped background.

A refinement graph of a model explicitly depicts all the type constraints expressed by the metamodels and models in the sense that whenever an edge is leading

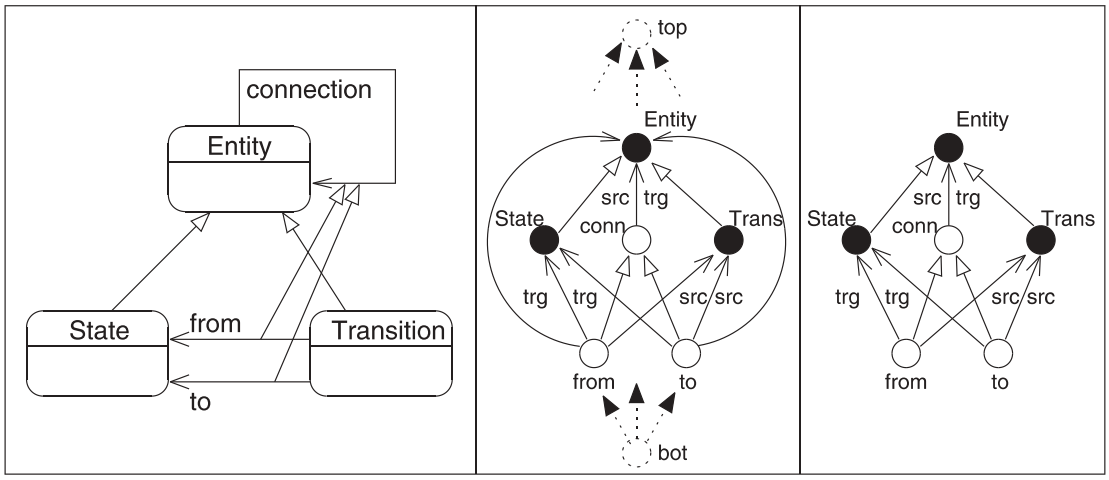


Fig. 12. Model graph and model lattice of transition systems

from a node n_1 to node n_2 we can deduce that n_2 is more abstract than n_1. In other words, the definition of model element n_2 must exist prior to introducing model element n_1.

5.2 The lattice of a refinement graph

In the following, we formally introduce two lattices representing (i) a single refinement graph and (ii) the *set* of refinement graphs to provide a formal analysis mechanism for type compliant evolution of models, e.g., to decide whether (i) a specific model is well-typed, (ii) and a model is a proper refinement of another (more abstract) model. Moreover, lower and upper bound operations will provide means to integrate different versions or different aspects of a model (e.g., created by different designers) into a safe and consistent global viewpoint of the system that is required by the advanced package merge constructs in the upcoming UML 2.0 standard.

Definition 10. *A lattice* $L(N) = (\sqsubseteq_N, \bot_N, \top_N, \sqcup_N, \sqcap_N)$ *is a five tuple where*

1. $\sqsubseteq_N$ *is a partial order on the set* N,
2. $\bot_N$ *is the infimum of* N, *that is* $\forall n \in N \bot_N \sqsubseteq_N n$
3. $\top_N$ *is the supremum of* N, *i.e.,* $\forall n \in N n \sqsubseteq_N \top_N$
4. $\sqcup_N$ *is the least upper bound of a set* $N_1 \subseteq N$ *is defined as* $\sqcup_N N_1 = u \iff \forall n \in N_1 : n \sqsubseteq u \wedge \forall u' \in N : n \sqsubseteq u' \supset u \sqsubseteq u'$.
5. $\sqcap_N$ *is the greatest lower bound of a set* $N_1 \subseteq N$ *is defined as* $\sqcap_N N_1 = u \iff \forall n \in N_1 : u \sqsubseteq n \wedge \forall u' \in N : u' \sqsubseteq n \supset u' \sqsubseteq u$.

The *least upper bound* $\sqcup_N$ of a set $N_1 \subseteq N$ is the least element that is larger than any elements in N_1. This can be determined by, for instance, a reachability analysis: (i) initially, the $\top_N$ element is added as singleton to the *current set* as it is an upper bound of all elements $n \in N_1$; (ii) all the predecessors of elements in the current set are tested whether they are still an upper bound of N_1. Those that satisfy this condition become the next *current set*, and this process is iterated until a fixpoint is reached.

The calculation of the *greatest lower bound* $\sqcap_N$ is similar, but this time one should start from the bottom element $\bot_N$ element and perform the dual steps.

As a direct consequence of its definition, one can show that the nodes of refinement graphs form a lattice.

Proposition 4. *The nodes of the refinement graph* $RG = (Nodes, Edges)$ *form a lattice* $L(Nodes) = (\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ *with* $\sqsubseteq = Edges$ *(the partial order of nodes is imposed by the edges),* $\bot = n_\bot$ *(the bottom extension node as infimum),* $\top = n_\top$ *(the top extension node as supremum), while* $\sqcup$ *and* $\sqcap$ *are defined as usual according to* $\sqsubseteq$.

As a result, many type conformance questions of a single model (like the following ones) can be answered directly on refinement graphs by the least upper bound and greatest lower bound operations of the lattice.

- *Is a model element A a supertype of model element B?* = the least upper bound of A and B is equal to B.
- *What is the common supertype of a set of model elements?* = the least upper bound of the set.
- *Does a link (model-level connection) correspond to its association (meta-level connection)?* = the least upper bound of link connection L, its source object entity O_{src}, the association connection A_{sup} of the link and the source class entity C_{src} is equal to C_{src} (due to the confluency of *instantiation* and *connection end* edges in the refinement graph);
- *Is there such a model element that is inherited (instantiated) from both model element A and B?* = the greatest lower bound of A and B is not equal to $\bot$.

Surprisingly, even more interesting results can be obtained if we establish another lattice for a *set* of refinement graphs.

5.3 The lattice of the sets of refinement graphs

In order to show that the set of refinement graphs also forms a lattice, we need to establish the notions of (1) the partial order relation on the set, (2) the infimum, and (3) the supremum of the set, and finally, (4) the least upper bound (lub) and (5) greatest lower bound (glb) operations.

Proposition 5. *Let $Set(RG)$ be the (finite) set of all finite refinement graphs (relevant for a specific purpose). The traditional subgraph relation is a partial order $\sqsubseteq$ on the set $Set(RG)$ of refinement graphs (which is an elementary result from graph theory).*

This time the calculation of greatest lower bound and least upper bound of a subset $X \subseteq Set(RG)$ needs some precautions in order to maintain the property that they both yield a well-formed refinement graph as the result.

Greatest lower bound. The calculation of the *greatest lower bound* $\sqcap X$ of a subset $X \subseteq Set(RG)$ *takes the intersection of all the refinement graphs* in X, and the *result is expected to be a common consistent abstract model* of all the models in X. For that reason, *at most* those nodes and edges are included in the result lattice that appear in all $x \in X$ lattices.

However, different models that share the same nodes may be in a conflict. Suppose that there is a refinement graph M_1 where $n_1 \longrightarrow n_2$ but in another model M_2, it is just the opposite $n_2 \longrightarrow n_1$. Since $\longrightarrow$ is a partial order, $n_1 \longrightarrow n_2 \wedge n_2 \longrightarrow n_1 \supset n_1 = n_2$, which also fulfills our expectations, as both configurations can be refined from an abstract model where the two nodes have not been distinguished yet. As there are no other refined models where these conditions could also be satisfied, we showed that by this method exactly the greatest lower bound of X is derived.

Note that the refinement graph consisting of the single $n_\top$ node is the infimum of the entire set $Set(RG)$. Thus we have the following proposition.

Proposition 6. *The set $Set(RG)$ of refinement graphs has an* ***infimum*** *$\bot_G$ (which is the graph consisting of the single node $n_\top$), and a* ***greatest lower bound*** *$\sqcap_G X = RG(Nodes, Edges)$ defined as*

1. *$\forall n \in Nodes : n \in x_1.Nodes \wedge \ldots x_n.Nodes, x_1, \ldots, x_n \in X$ (all the common nodes in the refinement graphs x_i are included)*
2. *$\forall n_1, n_2 \in Nodes : n_1 \longrightarrow n_2 \wedge n_2 \longrightarrow n_1 \supset n_1 = n_2$ (conflicting nodes are merged into a single node as a consequence of $\longrightarrow$ being a partial order)*
3. *$\forall e \in Edges : e \in x_1.Edges \wedge \cdots \wedge x_n.Edges, x_1, \ldots, x_n \in X \wedge e.from \in Nodes \wedge e.to \in Nodes$ (all the common edges are added that have both source and target nodes in $Nodes$)*

Corollary 1. *For all $G_1, G_2 \in Set(MG) : \sqcap_G\{G_1, G_2\} \sqsubseteq G_1$ and $\sqcap_G\{G_1, G_2\} \sqsubseteq G_2$.*

Informally, the greatest lower bound of two refinement graphs is a refinement of both of them.

Least upper bound. When calculating the *least upper bound* of some $X \subseteq Set(RG)$, one has to *take the union of all the graphs* $x \in X$, and, according to our informal expectations, this *union should be a refinement of all individual model graphs*. Unfortunately, in the case of contradicting models, this property cannot be established.

When taking the union of graphs $x \in X$, a merging is required along the nodes that appear in more than a single model. After that, the calculation of the union of the edges adds an edge from node n_i to node n_j if there is at least one model x_k with such an edge but there are no models with an edge leading to the opposite direction (i.e., from n_j to n_i). In the latter case, no edges are established in the result refinement graph between n_i and n_j in accordance with the properties of implication $n_i \longrightarrow n_j \vee n_j \longrightarrow n_i = \top$.

For the practical use, when a least upper bound of the set X is not a refinement of one or more models, the user can automatically return to the greatest consistent global state in the past by taking the greatest lower bound of X, which is inevitably a proper abstraction, and the refinement process can be redone from there in a controlled way.

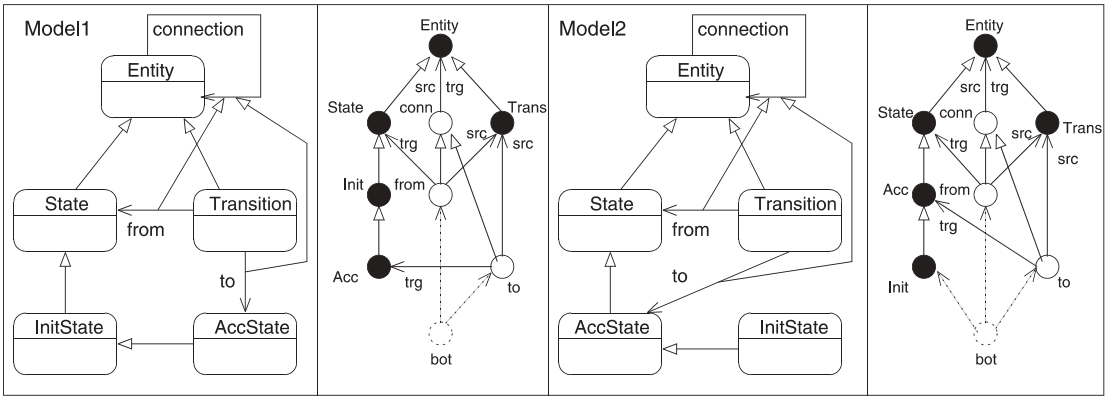
Proposition 7. *The set $Set(RG)$ of refinement graphs and a least upper bound $\sqcup_G X = M(Nodes, Edges)$ such that*

1. *$\forall n \in Nodes, x_1, \ldots, x_n \in X : n \in x_1.Nodes \vee \cdots \vee x_n.Nodes$ (all the common nodes)*
2. *$\forall e \in Edges, x_1, \ldots, x_n \in X : e \in x_1.Edges \vee \cdots \vee x_n.Edges \wedge e.from \in Nodes \wedge e.to \in Nodes$ (all the common edges)*
3. *$\forall n_1, n_2 \in Nodes : \neg(n_1 \longrightarrow n_2 \wedge n_2 \longrightarrow n_1)$ (conflicting edges are removed as a consequence of $\longrightarrow$ being a partial order)*

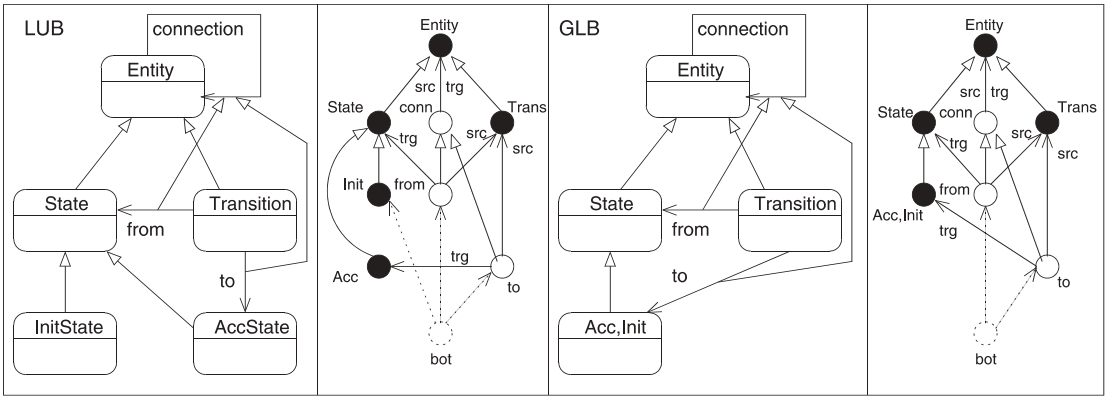
If the set $Set(RG)$ of refinement graphs is not controversial then we have a supremum $\top_G = \sqcup_G Set(RG)$, which is the least upper bound of all graphs in $Set(RG)$.

Example 2. In Fig. 13 the greatest lower bound and least upper bound of a set consisting of two models (introducing accepting and initial states in a deliberately contradicting way for finite automata) is calculated. For better understanding, the visual representations of the models are also depicted.

– In the case of the least upper bound (lub), the examination of the refinement graph detects that nodes *Acc* and *Init* are contradicting. For this reason, the *lub* supposes that both were introduced on purpose and keeps both of them, but the ordering relation between them is removed. As a result, we have an *AccState* and *InitState* derived by object inheritance from *State*, which is not a refinement of the two models (due to the contradicting inheritance).



(a) Two contradictory models



(b) The least upper bound greatest lower bound

Fig. 13. Calculating upper and lower bounds

– In the case of the greatest lower bound (glb), the examination of the refinement graph detects that nodes *Acc* and *Init* are equal. For this reason, they are treated as if all the edges entering one of them ends in this common state. As a result, we have an *AccState* derived by object inheritance from *State* but *InitState* must be reintroduced later in the design (or vice versa).

Proposition 8. *The set $Set(RG)$ of refinement graphs (ordered by the subgraph relation) forms a lattice $L_G = (\sqsubseteq_G, \bot_G, \top_G, \sqcup_G, \sqcap_G)$.*

Although we showed that the refinement graph structure of a single model is preserved when calculating lower and upper bounds of sets of refinement graphs, the result might be unexpected at the diagram level. For instance, if a connection (mapping) is redirected from one entity to another entity, the *intersection* of the two models might contain only the edge between the source entity and the connection (mapping) while the target of the connection (mapping) remains unspecified. However, this situation can easily be detected by comparing the degrees of each connection (mapping) node with the ones in the original refinement graph. Similarly, when taking the *union* of model representation, a connection (mapping) node may have additional outgoing edges, which fact can be detected similarly within the semantic domain. Both modeling contradictions are resolved automatically by our technique to a common consistent view as much as (theoretically) possible.

The final definition captures the notion of model (space) refinement in accordance with the previous results stating when a specific model is more detailed than another one.

Definition 11. *Let $M_1, M_2 \in Set(RG)$. If $M_1 \sqsubseteq_G M_2$ then M_1 is an **abstraction** of M_2, or conversely saying, M_2 is a **refinement** of M_1.*

5.4 Practical uses of model refinement

In Fig. 14, we demonstrate how some major concepts of object–oriented metamodeling can be captured formally by model refinement on our running examples of finite automata.

Structural extension. In a structural extension, a new model element is added to the model by refining (by in-

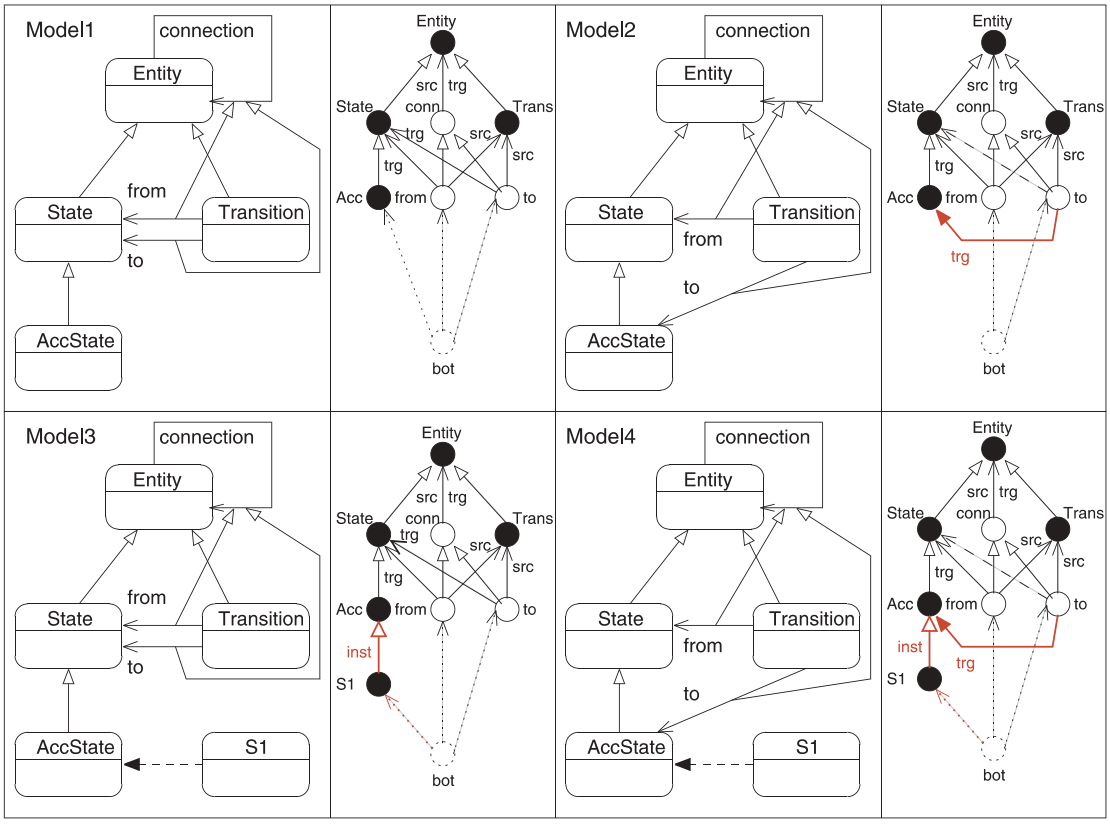


Fig. 14. Structural extension, relation restriction, instantiation

heritance or instantiation) an existing model element following our axioms.

For instance, a new entity (class) *AccState* has been derived by inheritance to the original diagram of Fig. 12 from the existing entity *State*. Although this time the designer can be quite certain that the refinement step he or she performed is correct, this can be verified formally on the semantic level by comparing the new refinement graph $Model_1$ to $Model_0$ of Fig. 12. One can conclude that the refinement step is correct as $Model_0 \sqsubseteq_G Model_1$ because node *Acc* and its incoming and outgoing edges were introduced by the refining operation.

Up to this point, the diagrams contained only the relationship on the class level. Now, when deriving $Model_3$ from $Model_1$ (and, similarly, $Model_4$ from $Model_2$), an instance *S1* of class *AccState* is introduced by refinement. This example demonstrates that classes and instances can be treated uniformly by the refinement graph: an instance is another node of the refinement graph derived from its class by instantiation. In many cases, both instantiation and inheritance holds between two entities (connections, mappings), which stems from the fact, mathematically, that a singleton (new element introduced as a leaf) only contains its own identifier.

Moreover, altering a model by the following set of operations turns out to be formally a model refinement in our sense (not proven here).

- Inserting an entity (e.g., a UML class) consistently as a leaf element or between two existing entities (refining the entity inheritance tree).
- Inserting an entity (e.g., a UML object) consistently as a leaf element into the instantiation hierarchy.
- Inserting a connection (e.g., a UML association or link) consistently (i.e., maintaining type constraints) as a leaf element or between two existing connection records (refining the connection inheritance tree).
- Inserting a mapping (e.g., a UML attribute or slot) consistently (i.e., maintaining type constraints) as a leaf element or between two existing connection records (refining the connection inheritance tree).
- Introducing multiple inheritance (or instantiation) for entities, connections or mappings (in which case the inheritance structure is no longer a tree but a directed acyclic graph).

On the one hand, several incorrectness properties (such as circularity in the inheritance structure, or invalid type refinements) can be detected on the refinement graph itself. In fact, they will never be introduced if the model refinement process is guided by the glb $\sqcap_G$ and lub $\sqcup_G$ operations.

On the other hand, the uniform refinement graph structure also allows a particular metamodeling approach to distinguish between classes and instances by introduc-

ing instantiation as a *connection* and thus deriving instances by *connection instantiation*.

Type restriction. In a type restriction, our model is not extended but altered by redirecting the refinement of an entity, connection or mapping. This could possibly mean that

- An entity refinement (either inheritance or instantiation) relation is established between two entities having previously the same parent in the inheritance structure. However, note that the inheritance of connections and mappings cannot always be redirected in this way as the proper inheritance of sources and targets may not be assured.
- The source (target) of a connection (or mapping) is redirected to a subtype of its former source (target) entity.

Our running example covers the latter case when $Model_2$ is generated from $Model_1$ (and, additionally, when $Model_4$ is derived from $Model_3$). In Fig. 14, the dashed lines can be derived by the transitive closure of solid lines; however, they are depicted explicitly to improve clarity of the subgraph relation when verifying that $Model_2$ is a proper refinement of $Model_1$.

"Future" concepts: redefine, import, package merge. The metamodeling concepts of the latest proposal for the upcoming UML 2.0 standard [38] are based upon three constructs (represented by dependencies with corresponding stereotypes) interpreted on packages (i.e., thus handling models and metamodels), namely, *redefine*, *import*, and *package merge*.

By the **redefine** operation (already well-known from many object-oriented programming languages that allow overriding of methods and properties), one can introduce a class derived from an existing one by inheritance with identical names but altered content in a new context (package). In a strict mathematical sense, the result of *redefine* should be a proper refinement of the redefined class. Therefore, as our VPM framework (and their refinement graph representation) does not depend on names, we can simply say that a redefine operation (if consistent) introduces new refinement (inheritance, to be precise) relations into the model space with identical names.

The **import** construct (that always leads between two packages) prescribes that all the contents of the source package are implicitly included in the target package. In our framework, the *import* construct is redundant in the sense that (i) on the one hand, a model element can be contained by multiple packages, (ii) on the other hand, refinement relations can "cross the borders of a package", thus there is no need to include them first in order to capture the intended refinement.

The **package merge** construct (demonstrated in Fig. 15) is probably the most complex metamodeling concept in the upcoming UML 2.0.

The overall idea is to derive metamodels (Package *C* in the figure) by merging existing ones (Package *A* and *B*)

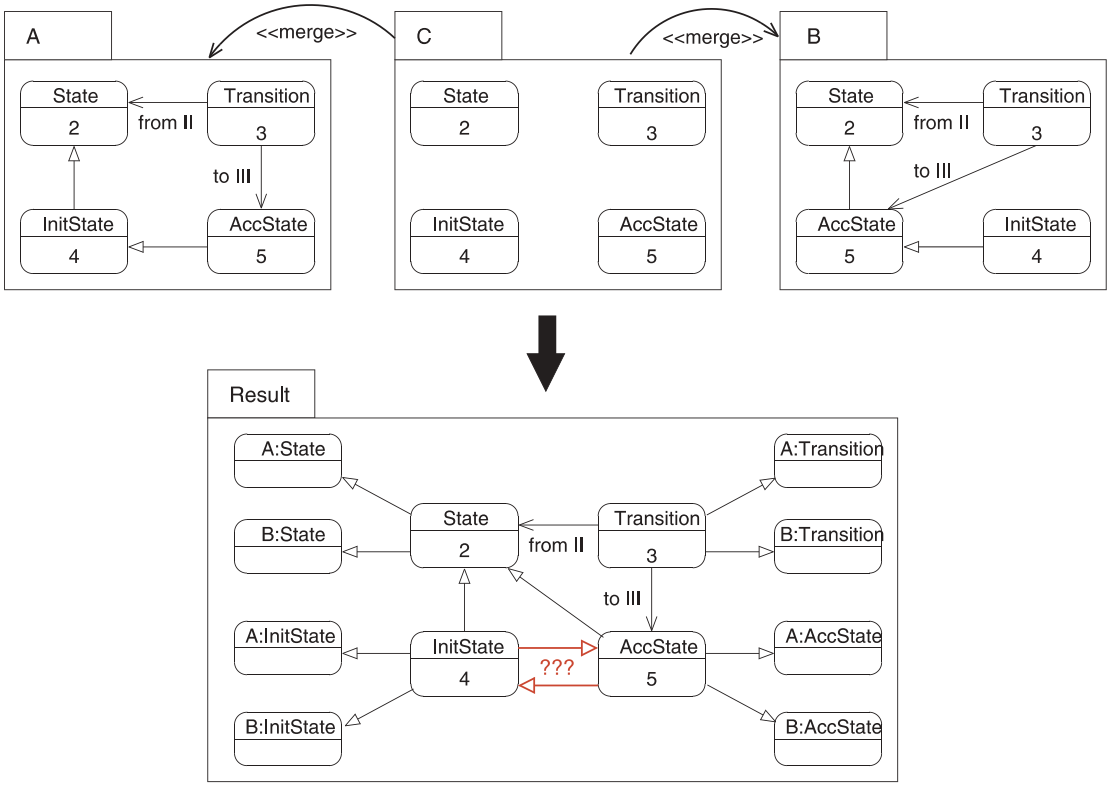


Fig. 15. Package merge

using the *redefine* and *import* operations in such a way that all the original associations and generalizations are also included in the result for the target package (Package *Result*).

- If two model elements with identical names appear both in the target and in a source package (for instance, see *State* in Package *A* and *C*), the package merge construct introduces a new generalization, and the class in the target package (*C:State*) is inherited from the class in the source (*A:State*).
- If such a model element is encountered that only appears in (at least) one of the source package but not in the target package (like the *from* connection in Package *A* or *B* but not in Package *C*), the model element is simply added.

However, the UML proposal [38] does not specify what happens (in the very common case) if the merge of multiple source packages results in inconsistencies (like the unclear intension of generalization between *AccState* and *InitState* in Fig. 15).

From a VPM point of view, package merges can be specified as multiple entity inheritance (in case of compound entities) between the source entities and the target entity. As a result of our static consistency analysis technique based upon the *lub* and *glb* calculation for refinement graphs, we can pinpoint such inconsistencies during package merge, furthermore, a consistent viewpoint (i.e., consistent from a refinement point of view and not from the viewpoint of package merge) is automatically obtained (see Fig. 13).

6 Case study: Uninterpreted modeling of UML designs

In the current section, we demonstrate the expressiveness of VPM on the application level by a complex case study that provides uninterpreted modeling facilities for UML designs. First, we briefly summarized the main ideas behind our uninterpreted modeling framework.

6.1 An introduction to uninterpreted modeling

Current formal methods fail to completely validate and verify detailed models of medium and large scale applications due to run-time and state space complexity limitations. Advanced analysis methods use a combination of different techniques to overcome these problems, like abstraction, exploitation of symmetries, result dependent model refinement, etc.

One of the standard approaches in abstraction based model analysis is *uninterpreted (or non-interpreted) modeling*. Fully interpreted models represent the flow of data through the system under evaluation by their appearance at the processing units and their respective timing and value. Non-interpreted abstraction reduces data representation to the appearance and temporal attributes of data at the nodes of the system and they omit value related information.

While this principle is most widely used in the context of data-flow networks, the most simple example is the decoloring of Petri-nets. A colored Petri-net, as a kind of interpreted model, can express both the dynamics and functionality of the modeled system by assigning color domains to the tokens corresponding to the type and domain, and by time stamps to the temporal attributes of the individual data. The initial marking and transition rules define the dynamics of the system. The abstraction leading to the uninterpreted model generates a non-colored Petri-net by keeping the structure of the system (all the places, transitions and edges between them) unaltered, but removing all the color domains from the model (place and transition domains, value dependent transition guards, etc.), thus substituting the colored tokens with uncolored ones.

- Obviously, this is a true abstraction in the sense, that every state sequence (trajectory) in the colored net has his counterpart in the uncolored net (a Galois connection is kept).
- On the other hand, the information loss originating in this abstraction may introduce spurious trajectories as well. For instance, the termination of a loop with a counter in a colored Petri-net model can be proven, but the elimination of the value depending guard in the loop branch simplifies the loop control to a random choice between the "*continue*" and "*exit*" branches, thus allowing even an infinite loop.

The uninterpreted model has a counterpart for each state transition sequence in the interpreted model, making uninterpreted modeling a favorite candidate for *semi-decision techniques* in validation and verification problems [30].

- If an exhaustive analysis performed on the uninterpreted model shows no violation of the validation objective, the user can be certain, that no problems will occur in the detailed, interpreted model. Note that the space and run-time complexities of an uninterpreted analysis is by orders of magnitude smaller than in the interpreted case due to the huge reduction of the state space by eliminating the rich domains associated with the different data types.
- However, a violation detected on the uninterpreted model may originate only from a spurious solution due to the abstraction, thus a detailed analysis performed on the uninterpreted model is required to filter out them.

6.2 Qualitative fault and error modeling

The assessment of dependability attributes necessitates the modeling of two basic phenomena:

- The local effects of *fault*s at the location, where they attack the system.

– The *propagation of errors* (fault induced wrong states of the system components) across the system in order to estimate, which faults may cause a *failure*, i.e. an observable deviation from the specified system behavior.

This way dependability analysis necessitates the simultaneous tracing of the information flow in the fault-free and in a faulty instance (or all candidate faulty instances) of the system in order to estimate where an observable difference in their behavior can be detected. The faithful mapping of the primary physical origins of faults to the model level necessitates a proper modeling of the underlying platform, and the deployment of the logic objects to physical (engineering) ones. [31] presents a model construction method to extend GRM [24] based models by the notion of errors and faults.

Qualitative fault modeling uses a small set of qualitative values from an enumerated type such as *good*, *faulty*, *illegal* (distinguishing between cases when (i) the data value is correct, (ii) syntactically correct, but different from the good value, or (iii) syntactically incorrect) to construct an abstract model reflecting the state of the resources and the potential propagation of errors. The designer can select this set of qualitative values arbitrarily according to the objective of the analysis. For instance, for timeliness analysis the set described above can be extended by the values of *early* and *late*.

The behavior of the system components may become non-deterministic due to uninterpreted modeling. For instance, a *faulty* value at one of the inputs of a multiplier propagates to the output, if the other input has a nonzero value, but this error propagation will be blocked by a zero on the other input. This way the transfer relation has to include both (*faulty,good,good*), (*faulty,good,faulty*) triples, as well.

The fault modeling methodology composes the model of two major parts:

– The qualitative model is constructed from the functional model by keeping all the objects and object references, but substituting the domain of every data type with the set of qualitative values.
– The model of each method is extended by the qualitative description of its reactions to an invocation having a faulty input value combination. By default, an action having a faulty value on at least one of its input pins reacts as randomly placing potentially illegal values (values randomly selected from the set {*good*, *faulty*, *illegal*}) at his outputs.

Naturally, the designer can override these defaults. For instance, the designer may exclude *illegal* values at the output of a fault-tolerant method.

Stateful methods execute their dynamics in an abstract form, where data dependent branches are transformed to random selections. The state space is extended by the failure state *illegal*, which models unhandled failures. Usually, this state should be unreachable in a proper design having a complete error exception handling system.

6.3 Uninterpreted modeling in VPM

An overview on how the previous uninterpreted modeling of UML designs can be captured within VPM in conformance with the four layer MOF architecture is provided in Fig. 16.

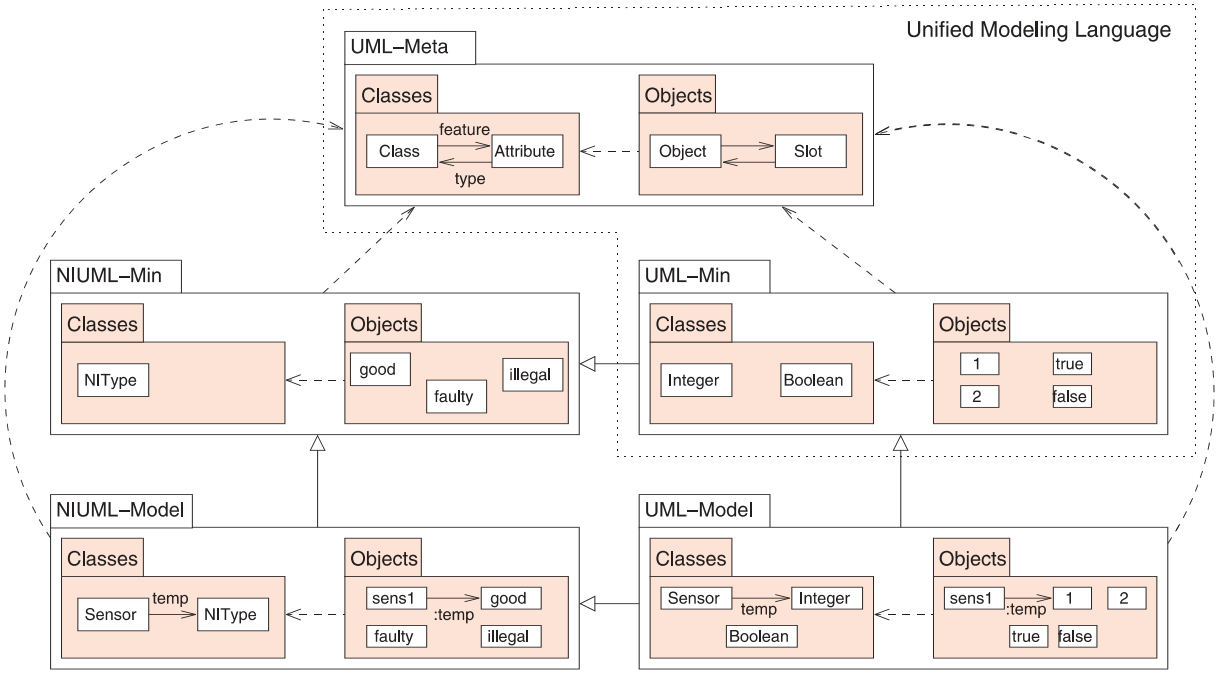


Fig. 16. Uninterpreted modeling of UML designs in VPM

- The metamodel of UML (*UML-Meta* on MOF level M2) serves as the underlying basis for both interpreted and uninterpreted modeling. The UML metamodel defines the traditional concepts on class and instances levels (like classes vs. objects, attributes vs. slots).
- All user models on level M1 (either interpreted or uninterpreted) are instances of the metamodel. However, the UML language definition also introduces some predefined types (and instances) like the class of *Integers*, and *Booleans* or the instances *1*, *2*, etc. that has to be implicitly included in any well-formed UML model. In this respect, the UML language definition not only contains the metamodel, but it also defines elements on the model (M1) level, which latter forms a minimal UML model (*UML-Min*).
- Given a specific UML design *UML-Model* (introducing, for instance, *Sensor*s having an *Integer* attribute for storing *temperature*), this M1-level model is, naturally, an instance of *UML-Meta*, but simultaneously, it is also inherited from the minimal UML model *UML-Min* in order to expressed that all constructs in the minimal UML model are included.
- In order to capture uninterpreted modeling of UML designs we introduce intermediate levels of abstraction between the UML metamodel *UML-Meta* and the model instance *UML-Model*.
  - First, we must introduce on the language level a class *NIType* (non-interpreted type) on level M1 (thus as an instance of *UML-Meta*) together with its corresponding instances (like *good*, *faulty* or *illegal*). This also forms a minimal non-interpreted core model *NIUML-Min* that must be contained by all user models. As an uninterpreted model should be more abstract than an interpreted one, we need to introduce refinement (inheritance) relations stating that *NIType* is a generalization of both *Integer* and *Boolean*, and all value instances (*1*, *false*, etc.) of these basic predefined types of UML are refinements of all uninterpreted values (like *good*, *faulty*).
  - Afterwards, the uninterpreted version of the user's UML model *NIUML-model* will be derived by inheritance from *NIUML-Min* (as each construct in *NIUML-Min* has to be refined or included) and by abstraction from *UML-Model* of in the meantime.

As a result, the generalization relations of interpreted and uninterpreted models are confluent in the sense additional levels of abstraction can be inserted at any point within the VPM hierarchy. Moreover, the case study also demonstrated that the language definition of UML is not situated merely on the M2 level, as it also encapsulates certain M1 level classes and instances that should be explicitly refined (contained) by any well-formed user model.

7 Conclusions

We presented a visual, and formally 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 and widespread reuse of both static structure and dynamic behavior. In addition, our static consistency analysis technique based on the notion of refinement graphs can detect and automatically resolve conflicting models and metamodels, thus providing a prime mathematical bases for appropriately handling the advanced modeling concepts (like package merges) of the upcoming UML 2.0 kernel language.

Note, however, that even though our metamodeling framework is much more concise (concerning the number of elements introduced as the kernel) and expressive (with dynamically reconfigurable metalevels), a metamodel alone cannot capture all the static well-formedness constraints of a modeling language. In the paper, we decided not to fix the way how domain engineers can specify additional static restrictions. On one hand, such constraints can be expressed by using OCL, in which sense our methodology is complimentary to existing techniques. On the other hand, a graph transformation rule without side effects (i.e., with identical left-hand side and right-hand side) can be interpreted as a static graphical constraint, in which case our rule refinement method provides in turn a certain level of reuseability.

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 [25]; GME [20], PROGRES [35], BOOM [28], or [21]) considers only a predefined number of metalevels. While only [4] (a framework for MML) and [5] 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 [4] suffers as identified in [5]).
- *VPM has a visual (UML-based) and mathematically precise specification language for dynamic behavior.* Like [14, 35] VPM uses a variant of graph transformation systems (introduced in [43]) for capturing the dynamic behavior of models, which provides a purely visual specification technique that fits well to a variety of engineering domains. However, the explicit inclusion of dynamic concepts in the metamodel is novel.
- *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 [9] and GME [20], 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 [11, 41, 43] in the field. Similar applications have been reported recently in [15, 16, 44].
– *VPM can precisely handle both engineering and mathematical domains* as demonstrated e.g., in [40] (for UML statecharts) and by the running examples of the paper (for Petri nets and finite automata).

The theoretical foundations introduced in the paper are supported by a prototype tool called VIATRA (VIsual Automated model TRAnsformations) [43]. 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 primarily aiming at to provide automated verification facilities for arbitrary model transformations within VPM. Our idea (following conceptually [10], which is a recent semantic framework for capturing transformations of programming languages) is to reason about the behavioral consistency of two models (let us call them source and target) taken possibly from different modeling languages by transforming their structure into a common abstract modeling language by graph transformation rules. Afterwards, the behavioral specification of the two models in the common representation should be derived automatically based upon (i) the operational semantics of the source and the target modeling language, and (ii) the model transformation rules. As a practical result, domain experts would obtain a framework for automated dynamic consistency between arbitrary dynamic UML diagrams (and profiles).

Additional future work is focused on an automated translation of mathematical structures (from formal definitions given in a MathML format) into their corresponding VPM metamodel in order to ease the handling of mathematical domains for transformation engineers.

Acknowledgements. The authors are grateful to Gergely Varró, John Rushby and many of his colleagues for their valuable comments.

References

1. Petri Net Markup Language. `http://www.informatik.hu-berlin.de/top/pnml`
2. Akehurst, D.: Model Translation: A UML-based specification technique and active implementation approach. Ph.D. thesis, University of Kent, Canterbury, 2000
3. Akehurst, D., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, LNCS, vol. 2460. Springer-Verlag, Dresden, Germany, 2002, pp. 243–258
4. Alvarez, J., Evans, A., Sammut, P.: Mapping between levels in the metamodel architecture. In: Gogolla, M., Kobryn, C. (eds.) Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools, LNCS, vol. 2185. Springer, 2001, pp. 34–46
5. Atkinson, C., Kühne, T.: The essence of multilevel metamodelling. In: Gogolla, M., Kobryn, C. (eds.) Proc. UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts and Tools, LNCS, vol. 2185. Springer, 2001, pp. 19–33
6. Atkinson, C., Kühne, T., Henderson-Sellers, B.: Stereotypical encounters of the third kind. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, LNCS, vol. 2460. Springer, Dresden, Germany, 2002, pp. 100–114
7. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability analysis in the early phases of UML based system design. International Journal of Computer Systems - Science & Engineering, 16(5): 265–275, 2001
8. Bondavalli, A., Majzik, I., Mura, I.: Automatic dependability analyses for supporting design decisions in UML. In: Proc. HASE'99: The 4th IEEE International Symposium on High Assurance Systems Engineering. 1999, pp. 64–71
9. Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation semantics for UML. In: Hussmann, H. (ed.) Proc. Fundamental Approaches to Software Engineering, FASE 2001 Genova, Italy, LNCS, vol. 2029. Springer, 2001, pp. 17–31
10. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, New York, NY, Portland, Oregon, 2002, pp. 178–190
11. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual automated transformations for formal verification and validation of UML models. In: Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering. IEEE Press, Edinburgh, UK, 2002, pp. 267–270
12. Csertán, G., Pataricza, A., Harang, P., Dobán, O., Biros, G., Dancsecz, A., Friedler, F.: BPM based robust E-Business application development. In: Proc EDCC-4 Fourth European Dependable Computing Conference, LNCS, vol. 2485. Springer, Toulouse, France, 2002, pp. 32–43
13. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific, 1999
14. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 – The Unified Modeling Language. Advancing the Standard, LNCS, vol. 1939. Springer, 2000, pp. 323–337
15. Engels, G., Heckel, R., Küster, J.M.: Rule-based specification of behavioral consistency based on the UML meta-model. In: Gogolla, M., Kobryn, C. (eds.) UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools, LNCS, vol. 2185. Springer, 2001, pp. 272–286
16. Heckel, R., Küster, J., Taentzer, G.: Towards automatic translation of UML models into semantic domains. In: Proc. AGT 2002: Workshop on Applied Graph Transformation. Grenoble, France, 2002, pp. 11–21
17. Huszerl, G., Majzik, I.: Quantitative analysis of dependability critical systems based on UML statechart models. In: HASE 2000, Fifth IEEE International Symposium on High Assurance Systems Engineering. 2000, pp. 83–92
18. Kobryn, C.: UML 2001: A standardization Odyssey. Communications of the ACM, 42(10), 1999
19. Latella, D., Majzik, I., Massink, M.: Automatic verification of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing, 11(6): 637–664, 1999

20. Ledeczi, A.:, Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: Proc. Workshop on Intelligent Signal Processing. 2001
21. Naumenko, A., Wegmann, A.: A metamodel for the unified modeling language. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, LNCS, vol. 2460. Springer, Dresden, Germany, 2002, pp. 2–17
22. Object Management Group. Software Process Engineering Metamodel (SPEM). http://www.omg.org
23. Object Management Group. UML Profile for Enterprise Distributed Object Computing (EDOC). http://www.omg.org
24. Object Management Group. UML Profile for Schedulability, Performance and Time. http://www.omg.org
25. Object Management Group. Meta Object Facility Version 1.3, 1999. http://www.omg.org
26. Object Management Group. Model Driven Architecture – A Technical Perspective, 2001. http://www.omg.org
27. Object Management Group. Object Constraint Language Specification (in UML 1.4), 2001. http://www.omg.org
28. Övergaard, G.: Formal specification of object-oriented metamodelling. In: Maibaum, T. (ed.) Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany, LNCS, vol. 1783. Springer, 2000
29. Pap, Z., Majzik, I., Pataricza, A.: Checking general safety criteria on UML statecharts. In: Voges, U. (ed.) Computer Safety, Reliability and Security (Proc. 20th Int. Conf., SAFECOMP-2001), LNCS, vol. 2187. Springer, 2001, pp. 46–55
30. Pataricza, A.: Semi-decisions in the validation of dependable systems. In: Suppl. Proc. DSN 2001: The International IEEE Conference on Dependable Systems and Networks. Göteborg, Sweden, 2001, pp. 114–115
31. Pataricza, A.: From the general resource model to a general fault modeling paradigm? In: Workshop on Critical Systems Development with UML at UML 2002. Dresden, Germany, 2002
32. Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations: Foundations. World Scientific, 1997
33. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999
34. Schürr, A., Sim, S.E., Holt, R., Winter, A.: The GXL Graph eXchange Language. http://www.gupro.de/GXL/
35. Schürr, A., Winter, A.J., Zündorf, A.: In: [13], chap. The PROGRES Approach: Language and Environment. World Scientific, 1999, pp. 487–550
36. Singh, A., Billington, J.: A formal service specification for IIOP based on ISO/IEC 14752. In: Jacobs, B., Rensink, A. (eds.) Proc. Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002). Kluwer, Enschede, The Netherlands, 2002, pp. 111–126
37. Taentzer, G.: Towards common exchange formats for graphs and graph transformation systems. In: Padberg, J. (ed.) UNIGRA 2001: Uniform Approaches to Graphical Process Specification Techniques, ENTCS, vol. 44(4). 2001
38. U2-Partners. UML: Infrastructure v. 2.0 (Third revised proposal), 2003. http://www.u2-partners.org/artifacts.htm
39. Varró, D.: Automatic program generation for and by model transformation systems. In: Kreowski, H.-J., Knirsch, P. (eds.) Proc. AGT 2002: Workshop on Applied Graph Transformation. Grenoble, France, 2002, pp. 161–173
40. Varró, D.: A formal semantics of UML Statecharts by model transition systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Proc. ICGT 2002: 1st International Conference on Graph Transformation, LNCS, vol. 2505. Springer-Verlag, Barcelona, Spain, 2002, pp. 378–392
41. Varró, D., Gyapay, S., Pataricza, A.: Automatic transformation of UML models for system verification. In: Aranjo, J., Whittle, J., Toval, A., France, R., Moreira, A. (eds.) WTUML'01: Workshop on Transformations in UML. Genova, Italy, 2001, pp. 123–127
42. Varró, D., Pataricza, A.: Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, LNCS, vol. 2460. Springer-Verlag, Dresden, Germany, 2002, pp. 18–33
43. Varró, D., G. Varró, Pataricza, A.: Designing the automatic transformation of visual languages. Science of Computer Programming, 44(2): 205–227, 2002
44. Whittle, J.: Transformations and software modeling languages: Automating transformations in UML. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications, LNCS, vol. 2460. Springer-Verlag, Dresden, Germany, 2002, pp. 227–242
45. World Wide Web Consortium. MathML 2.0. http://www.w3c.org/Math

András Pataricza is an associate professor at the Department of Measurement and Information Systems of Budapest University of Technology and Economics. He lead several national and international research projects in the field of transformation based analysis of UML models. He held tutorials on the topic of UML and dependability at Safecomp 2001, UML 2002, and at the IEEE Dependable Systems and Networks Conference (DSN 2003). He was the leader of the IBM-sponsored project to create an E-Business Academy curriculum, which is currently used by more than 30 higher education institutions.

Dániel Varró was graduated at the Budapest University of Technology and Economics, and currently, he is a final year PhD student at the Department of Measurement and Information Systems. His main research interest is to design automated and provenly correct model transformations within and between visual modeling languages with a special focus on UML. He was a former visiting researcher at SRI International and at the University of Paderborn.