

## Some Preliminary Hints on Formalizing UML with Object Petri Nets

Luciano Baresi

Dipartimento di Elettronica e Informazione – Politecnico di Milano  
Piazza L. da Vinci 32, I-20133 Milano, Italy  
*bares@elet.polimi.it*

**ABSTRACT:** Petri nets have already been used to formalize UML and they have already shown – at least partially – what can be done in terms of analysis and simulation. Nevertheless “conventional” Petri nets, like P/T nets and color nets, are not always enough to efficiently formalize the behavior associated with UML models when specifications heavily rely on typical object-oriented features, like inheritance, polymorphism and late binding. Rendering these peculiar aspects by means of ad-hoc subnets oftentimes lead to awkward subnets and impacts the complexity of resulting formalizations.

This paper presents some preliminary hints on formalizing the dynamic aspects of UML – more precisely Interaction and Statechart diagrams – with Object Petri Nets (OPNs) to find a more natural solution to the formalization of object-oriented features. These advanced nets conjugate the capabilities specific to object-oriented systems with those that usually belong to Petri nets. The adoption of OPNs has both advantages and disadvantages: embedded characteristics help formalize the most challenging concepts, but at the same time defined nets lose simplicity and their meaning is not always clear at the first glance.

Besides exemplifying some common problems, like inheritance and late-binding, the paper compares OPNs and high-level nets, as to the support offered to formalize UML, and proposes a (semi) automatic approach for making the translation process automatic.

### I. INTRODUCTION

Nowadays, UML is the OMG – but also the de-facto – standard for object oriented analysis and design. Its success comes from several different factors, but among them its simplicity plays a dominant role. The capability of using the language after a few-hour training and the possibility of being proficient in it without any strong mathematical background made industry accept and exploit UML ([6]).

Even if the simplicity and graphical syntax of UML are two key characteristics, its informality – or partial formality – is a barrier against more thorough uses of designed models. They are mainly documentation that is used throughout the different production phases, but cannot be used for automatic analyses, simulation, and test case generation. These problems do not motivate the adoption of more formal methodologies like Petri nets [17], VDM [11], or in

these days Alloy [9], but claim for approaches that make UML more precise and thus suitable for automatic analysis and validation. In order to benefit from a simple graphical notation, we cannot change the concrete syntax, but we have as much room as we want to work and improve its semantics and provide the user with a *dual* approach ([1]): He uses “standard” UML to model the problem and identify possible solutions, but the validation/verification engine works on a more precise (formal) representation of models that, hidden to the user, supplies the needed degree of rigor and possibly the automated verification tools.

Since UML is a wide notation, the formalization of all notation aspects and elements with a single formal model is almost unfeasible. Seminal papers (for example [22]) have already taught us that different formal methods can be employed to render the different aspects of the same models; the same approach can be used to formalize UML: Interesting aspects can be captured with ad-hoc formal languages.

In this paper, we concentrate on the dynamic aspects and we propose Petri nets as the formal means to ascribe UML – more precisely Interaction and Statechart diagrams – with formal semantics. The author has already described the first experiments on formalizing UML with high-level Nets (HLPNs) [3], [4], but obtained results were twofold: Encouraging and interesting because of the analysis capabilities; discouraging and unsatisfactory as to the capabilities of modeling the key features of object-oriented systems and the constraints on transformable UML models.

Trying to improve negative outcomes, we decided to investigate the use of an extension to Petri nets. Among the different alternatives of object-flavored nets, we decided to do some experiments with Object Petri nets by Lakos ([14], [13]) because they embed almost all object-oriented features and can be mapped directly onto behaviorally equivalent colored Petri nets [10].

The paper uses a simple case study to present how OPNs can be used to ascribe formal semantics to UML. More specifically, we address its peculiar features, and compare obtained results with respect to what would be possible with HLPNs. The paper sketches also a framework to make the translation process automatic and let the user benefit from the OPN semantics.

The rest of this paper is organized as follows. Section II briefly introduces Object Petri Nets and Section III de-

scribes how they can be used to render UML elements. Section IV compares OPNs and HLPNs in terms of the support they offer to “formalize” UML. Section V presents an hypothetical framework, based on previous experiences to make the transformation process automatic. Section VI briefly surveys similar proposals and Section VII concludes the paper.

## II. OBJECT PETRI NETS

*Object Petri Nets* [14], [13], [12], OPNs for short, are one of the most interesting object-oriented extensions to Petri nets. They supply the “right” mix between places and transitions, that is, the conventional Petri nets’ graphical representation, and object oriented concepts.

In OPNs, *classes* are rendered as subnets that can be instantiated as many times as needed to obtain *objects*. This is the first important feature since it is the way OPNs support the dynamic creation of objects. Each class/subnet can contain *data fields*, *functions*, and *transitions*. Data fields, which generalize PN places, can be values of some primitive types (e.g., integer, real, boolean), instances of other classes, or multisets of the two previous categories (i.e., “real” PN places). This way, classes can be instantiated as if they were tokens, which could embed arbitrary subnets.

Each field can also have an initial value, i.e., an initial marking, and a guard that must always hold true: The place marking must always satisfy the predicate. Both functions and transitions have bindings with variables and are evaluated at particular points in time. The former are read-only computations on values, i.e., they do not consume “tokens”; the latter modify the net’s state according to the usual PN semantics: “tokens” are removed from the preset and produced in the postset. Functions are *state dependent* if they are associated with specific places and must be evaluated against the contents of these places.

Data fields and functions can be either *local* (private) or *exportable* (public). Fields can also be *local to transitions*, i.e., their values exist only for the duration of transitions’ firings. *Super classes* and *super transitions* extend places and transitions by embedding complete subnets into them. A *super place* is defined by a class that inherits a multiset type, defines the tokens that can be added and removed, and – if needed – the ways this can be done by means of a subnet. A *super transition* is a way of collapsing a complete lower-level subnet into a single higher-level transition: Atomicity is preserved, but it is obtained through a set of “hidden” steps.

Classes can *inherit* properties from other classes, with the convention that, for the sake of readability, inherited properties are always repeated in the classes that inherit them. OPNs adopt the Eiffel ([16]) convention that the type of an overriding field must be a subclass of the type of the overridden field and that the guards in the subclass must always be stronger than those in the superclass. Inheritance and dynamic instantiation pave the ground to *dynamic binding*.

Tokens (instances) of a given subclass can always be used in place of tokens of the corresponding superclass.

As an example, Figure 1 presents two classes for the *Russian Philosophers problem*, one of the many variations on the dining philosophers problem, taken from [12]. As convention, inherited elements are depicted in gray and exported properties with a double outline.

Moving to the excerpts of the example, the class *RPhil* (Figure 1(a)) defines Russian philosophers as subclass of class *DPhil*, that is base philosophers. The subclass inherits *n*, the number of philosophers, *id*, the philosopher’s identifier, and the two functions *left()* and *right()*, to let each philosopher identify who is seated by him; it redefines *hungry()* and defines *image* as an element of class *DTable*.

The class *RTable* (Figure 1(b)) is not a “pure” data structure, like *RPhil*, and comes with a proper subnet to define its behavior. The meaning of each transition is fully explained by the dynamics of the philosophers problem, but we have to notice that this net allows for dynamic binding. In fact, all places defined as *DPhil\**, that is multisets of *DPhil* philosophers, could contain also *RPhil* philosophers. Only, transition *retRight* has been redefined to “produce” *RPhil* tokens; all other transitions have been inherited from *DTable*. Finally, just to point out the graphical convention, function *dead()* must be evaluated on the contents of place *hasLeft*.

## III. FROM UML TO OPNS

The first experiments with OPNs demonstrated the suitability of the formal methods and suggested also some draft heuristics to transform UML models into OPNs. In a given sense, the translation is more scoped by the fact that each class is rendered using a particular OPN subnet:

- Simple data classes (i.e., abstract data types) are rendered with a set of data entries and functions, but without any real net to describe their behavior.
- Classes that supply *services* are rendered with super classes to be able to represent both requests and provisions of services as arcs entering/leaving the super place. Even if they are not associated with any *UML statechart diagram*, they are rendered with a simple – single state – Petri net to be able to accept and serve outside requests.
- Specific behaviors, in the form of UML statechart diagrams, are rendered with ad-hoc Petri nets.
- UML associations are rendered using pairs of places, whose actual types depend on the classes they connect and the multiplicity declared for the association end.
- Inheritance, polymorphism, and late binding are rendered directly using OPN features, as exemplified in Section II.
- UML interaction diagrams can only be used to properly set the textual annotations associated with the elements in each class (subnet). They do not define interclass PN arcs.

A simple example on which we can apply the heuristics listed so far, is presented in Figure 2. The class diagram

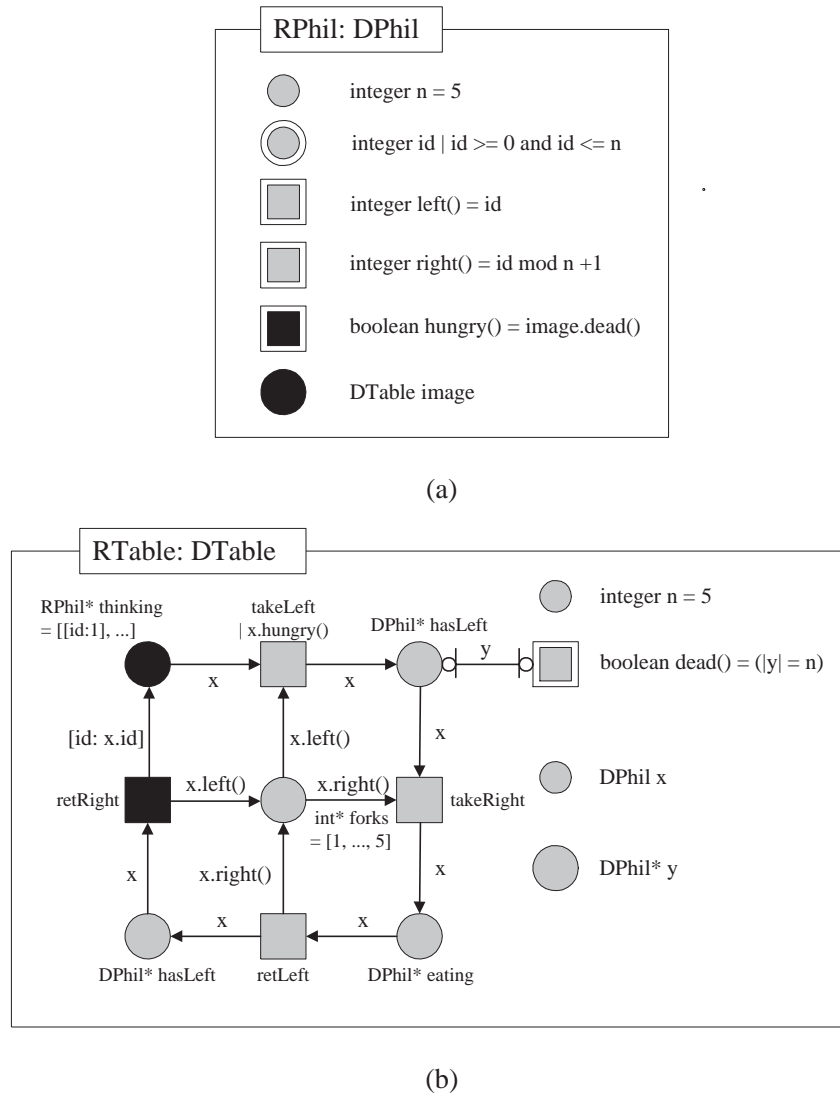


Fig. 1. Excerpts of the OPN solution to the *Russian Philosophers problem* from [12]

(Figure 2(a)) describe a *Tool* that can operate on two different kinds of pieces: *2Dpieces* and *3Dpieces*, respectively, where the second class inherits from the first one. The tool, while modeling the different pieces shows also the behavior described by the statecharts of Figure 2(b), which basically imposes a further modeling step for *3Dpiece* objects.

Given the rules (heuristics) defined so far, the simple problem can be formalized by the OPNs of Figure 3. The transformation of the two *Piece* classes is simple and straightforward: Each attribute becomes a data entry and each method becomes a function. Notice that all properties are inherited, but the function *type()* is redefined. A bit more complex is the transformation of the *Tool* class: The super place accepts invocation of the *model* method and we see both incoming and outgoing arcs with  $p$  as annotation to translate the method invocations and the results, respectively. The net that defines the internal semantics of

the class mimics the statecharts associated with the UML class: PN places are used to render states and PN transitions to represent state transitions. The entry and exit points are in charge of accepting incoming requests and supplying results, respectively.

#### IV. OPNs vs. HLPNs

Given the simple examples in the previous sections, and given the previous author's experiences ([3], [4]) with HLPNs, we can summarize these first experiments as follows:

1. The topology of resulting net(s) is completely different. With HLPNs, we have one complete net, which embed all the subnets that correspond to the different classes/elements. With OPNs, we have a set of subnets, one for each class, but we miss the big picture. All the connections among nets are not clearly stated by the topology,

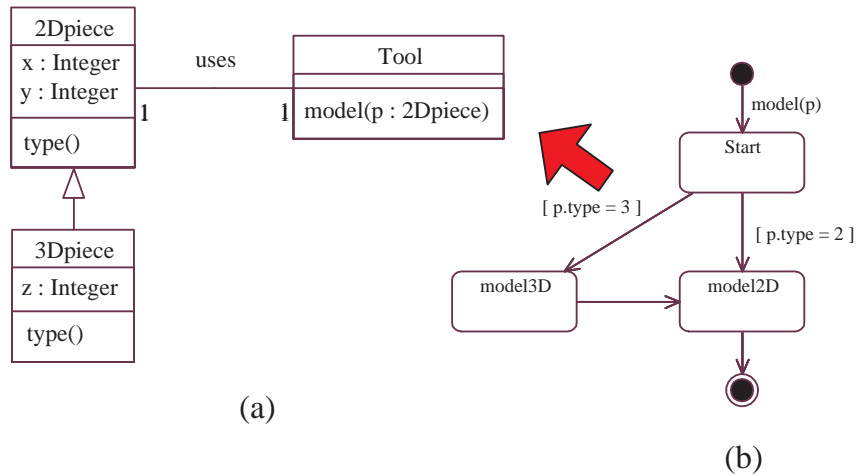


Fig. 2. A simple UML example

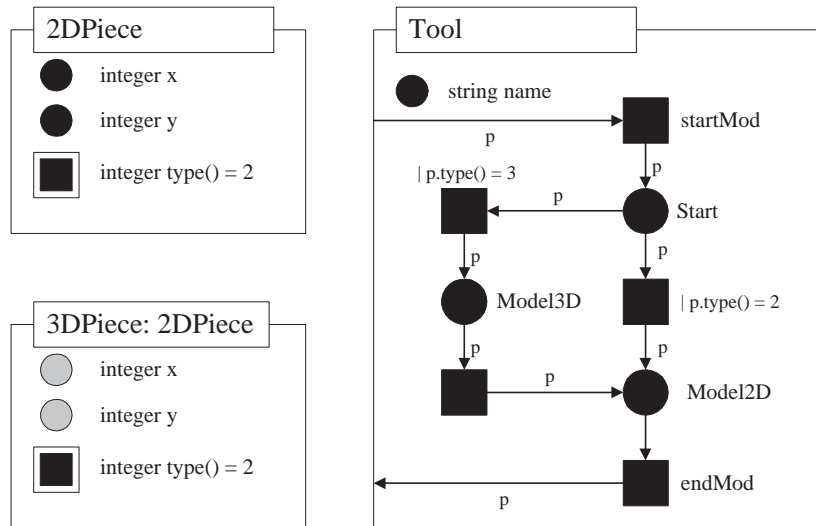


Fig. 3. The corresponding OPN subnets

but have to be “understood” using the textual annotations associated with each subnet.

2. The previous consideration implies also that the interaction exemplified by *UML interaction diagrams* cannot be rendered with OPNs: They remain hidden in the textual annotations associated with class subnets. HLPNs do not show the same flexibility, but again would use interaction diagrams to complete the connections among the different subclasses.

3. The rendering of tokens is different. With HLPNs objects can only be tokens in the net. With OPNs, objects are different instantiations of the subnets that correspond to classes. Again, this means that in the first case, the net is always the same and what changes is the marking, while in the second case, the topology – if we can identify it – evolves with the system.

4. Needless to say, specific object-oriented features, which could only be mimicked with HLPNs, can simply and straightfully be rendered with OPNs. The main difference is that with HLPNs, these constructs can only be implemented using “strange” topologies and relying on the programming language used to annotate the nets.

5. The key consideration applies to analysis capabilities. We could say that the two approaches have pros and cons that could make the user decide for either approach, but when we move to analysis, the scenario is a bit different. OPNs, even if can be mapped onto behaviorally equivalent colored Petri nets, do not provide special-purpose analysis tools, since to the best of author’s knowledge they are still under implementation. If we moved to HLPN, we can say that since we ask for less to the analysis tools, we can better reuse available systems (for example, Cabernet [19]).

## V. FORMALIZATION FRAMEWORK

Even if the supporting tools are not as good as we would like, for sure OPNs provide interesting features to ascribe UML dynamic aspects with formal semantics. The *dual* approach proposed in this paper is based on *MetaEnv* ([1], [2]), a meta CASE tool that allows for close integration between graphical specification notations, UML in this case, and formal engines, OPNs in this paper. The environment is based on rules, mainly graph grammar productions, to transform graphical models (UML models) into representations that can more easily be simulated and animated. In this paper, we do not want to concentrate on *transformation rules* and their technical details, but we want only to sketch the main components of *MetaEnv* that could be reused for transforming UML into behaviorally equivalent OPNs.

The hypothetical supporting framework is organized around a three-tiered architecture (Figure 4). Its components can easily be split in *concrete*, *abstract*, and *semantic* components according to the level at which they work.

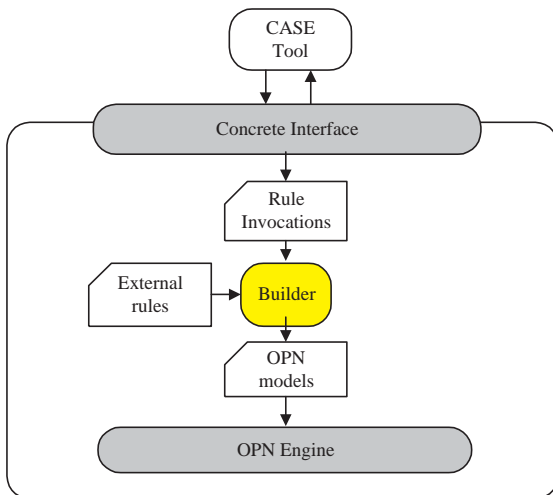


Fig. 4. Architecture of *MetaEnv*

Concrete components are the *CASE Tool*, for example *Rational Rose*, and its *interface* to export produced artifacts. Nowadays, a simple solution to implement this interface would be through XMI (XML Metadata Interchange, [18]) which allows all compliant tools to export their artifacts as XML documents. If we wanted a two-way interface able not only to export artifacts, but also to import external results, we should exploit tool-specific features. Differently from all other *MetaEnv* components, the concrete interface varies according to the used CASE tool.

The only abstract components is a *Builder*, which is a graph grammar interpreter that applies transformation rules according to the application sequence defined by the concrete interface and builds all the OPNs. The *Builder* read its rules from an external repository that contains all required rules. Besides “standard” transformation rules, users

can define and add the rules they prefer, and thus transform UML models into OPNs according to their ideas.

The semantic component should be the *OPN Engine*. It defines, executes and analyzes the OPNs obtained through the builder. The layered and component-based architecture of *MetaEnv* allows for experiments with different formal engines without rebuilding the system from scratch. The interfaces and the services provided by/required from the engine are clearly stated and thus plugging in a new engine is quite easy.

The described organization fosters a one-way transformation, that is from UML to OPNs. A more complete approach would require a two-way translation, that is also the transformation of results produced by the OPN engine into visualizations/animations on the UML model. *MetaEnv* would support also this second transformation, but to supply significant results we would need to further study the analysis capabilities and this is out of the scope of this paper.

## VI. RELATED WORK

As already pointed out in the Introduction, the degree of precision and rigor of UML is currently seen as a problem and several different proposals are already available with possible solutions.

As to other approaches that formalize UML with different flavors of object oriented Petri nets, we can mention the work by Saldhana and Shatz [21], who propose an approach to transform UML Statecharts and Collaboration diagrams into Colored Petri nets through *object Petri net models*. Their work does not address UML classes as the starting point to derive the properties of the system and propose a fixed and one-way mapping that exploit the intermediate means, the object Petri models, to produce colored Petri nets. Another interesting proposal is by Philippi ([20]), who intertwines UML and Petri nets to propose a *seamless object-oriented software development process*. The key goal is a kind of integration between UML class diagrams and OOPr/T models, yet another extension of Pr/T nets with object-oriented features. The interesting aspect is that the approach aims to automatically produce Java code.

If we moved to other ways of making UML more precise, or other languages to do that, we can mention vUML [15] that translates ad-hoc UML statecharts into PROMELA, the input language of the SPIN model checker [7]. This proposal, even if interesting, is only partial since it considers only state diagrams and the limitations and constraints are heavy. We must also mention the work at IFAD [8] to sell an interesting mapper between VDM++ [5] and UML: Here the limitations come from the iterative nature of the proposed modeling solution. VDM++ specifications can be rendered in UML, but also special-purpose UML specifications – mainly stereotyped class diagrams – must be convertible in VDM++.

## VII. CONCLUSIONS AND FUTURE WORK

The paper presents some preliminary hints on ascribing formal semantics to some dynamic aspects of UML through OPNs (Object Petri Nets). Besides showing some example nets to make the reader understand what OPNs are and what he can do, it shows a simple example and compares OPNs with HLN as to modeling features and observability of results. These first experiments gave encouraging answers, but the approach and the translation process need to be further refined. We still have to find the right compromise between a simple and straightforward translation process and the degree of freedom left to the UML user while designing his models. Moreover, we will work on assessing the actual analysis features offered by OPNs – unfortunately today are only partially implemented – and understanding how they can be used to offer significant results to UML users.

Moving a step further, this proposal will be part of a more general and complex approach that will be *dual* and *multi-language* to ascribe formal semantics to the whole UML, allowing users to exploit the formal view they need/prefer for the particular problem.

### REFERENCES

- [1] L. Baresi. *Formal Customization of Graphical Notations*. PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. in Italian.
- [2] L. Baresi, A. Orso, and M. Pezzè. Introducing Formal Methods in Industrial Practice. In *Proceedings of the 19th International Conference on Software Engineering*, pages 56–66. ACM Press, 1997.
- [3] L. Baresi and M. Pezzè. On Formalizing UML with High-Level Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets*, pages 276–304. Springer-Verlag, 2001.
- [4] L. Baresi and M. Pezzè. Improving UML with Petri nets. In *Proceedings of ETAPS2001 Workshop on Uniform Approaches to Graphical Process Specification Techniques (UniGra)*, March 2001.
- [5] E. H. Dürr and N. Plat. VDM++ Language Reference Manual. Technical report, IFAD - The Institute of Applied Computer Science, 1995.
- [6] M. Fowler. *UML Distilled*. Addison-Wesley, 1997.
- [7] G. J. Holzmann. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–95, May 1997.
- [8] IFAD - Denmark. *VDM++ User's Manuals*, 1998.
- [9] D. Jackson. Alloy: A Lightweight Object Modelling Notation. [sdg.lcs.mit.edu/~{ }dnj/pubs/alloy-journal.pdf](http://sdg.lcs.mit.edu/~{ }dnj/pubs/alloy-journal.pdf), July 2000.
- [10] K. Jensen. Coloured Petri Nets. In W. Reisig and G. Rozemberg, editors, *Advances in Petri Nets*, volume 254-255 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin-New York, 1987.
- [11] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- [12] C. A. Lakos. Object-Oriented Modeling with Object Petri Nets. In *Concurrent Object-Oriented Programming and Petri Nets*, pages 1–37. Springer-Verlag, 2001.
- [13] C. A. Lakos. Object Petri Nets — Definition and Relationship to Coloured Nets. Technical Report R94-3, Department of Computer Science, University of Tasmania, April 1994.
- [14] C. A. Lakos. The Object Orientation of Object Petri Nets. In *Proceedings of the first international workshop on Object-Oriented Programming and Models of Concurrency - 16th International Conference on Application and Theory of Petri Nets*, pages 1–14, June 1995.
- [15] J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. In *14th IEEE International Conference on Automated Software Engineering*, pages 255–258. IEEE Computer Society Press, 1999.
- [16] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.
- [17] T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
- [18] Object Management Group. Xml metadata interchange (XMI) specification. Technical report, OMG, November 2000.
- [19] M. Pezzè and S. Silva. Cabernet User Manual. Technical Report 47-94, Politecnico di Milano, May 1994.
- [20] S. Philippi. Seamless Object-Oriented Software Development on a Formal Base. In *Proceedings of Workshop on Software Engineering and Petri-Nets, held at the 21st International Conference on Application and Theory of Petri-Nets*, June 2000.
- [21] J. Saldhana and S. M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 103–110, July 2000.
- [22] P. Zave and M. Jackson. Where Do Operations Come From? A Multiparadigm Specification Technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, 1996.