Improving UML with Petri nets [*]

Luciano Baresi

*Dipartimento di Elettronica e Informazione, Politecnico di Milano
piazza Leonardo da Vinci, 32. I-20133 Milano, Italy* [1]

Mauro Pezzè

*Dipartimento di Informatica, Automatica e Comunicazioni,
Università degli Studi di Milano Bicocca,
via Bicocca degli Arcimboldi, 8. I-20126 Milano, Italy* [2]

Abstract

UML is the OMG standard notation for object-oriented modeling. It is easy, graphical and appealing, but in several cases still too imprecise. UML is strong as modeling means, supplies several different diagrammatic notations for representing the different aspects of a system under development, but lacks simulation and verifiability capabilities. This drawback comes from its semi-formal nature: UML is extremely precise and wide if we consider syntactical aspects, but its semantics is as precise as those of informal notations. Scientists and users, together with standardization efforts (UML 2.0), are trying to overcome this problem, but as side effect, they are also limiting the intrinsic flexibility of UML. Moreover, several formalization efforts concentrated on its static elements (for example, inheritance), leaving dynamic semantics almost untouched.

In this paper we propose the paring of UML dynamic models with high-level timed Petri nets (HLTPN) to obtain a flexible and customizable means to reason on the dynamic aspects of object-oriented models, to simulate particular parts of these models, and if necessary analyze them. The proposal exploits rules to ascribe main UML elements with formal semantics in terms of functionally equivalent HLTPNs and to show results (from execution and analysis) as decorations to UML symbols. Besides sketching the approach, the paper presents also some experiences we have gained so far with it and a research agenda to identify other possible uses of the dual definition of the notation.

Key words: UML; Petri Nets; Formal Specifications; Simulation; Analysis

1   Introduction

Informal methods are still ahead in the competition with formal ones. Syntactic richness, user friendliness, simplicity, and flexibility of informal methods win over strong simulation and analysis capabilities offered by formal ones [17,11].

During requirements elicitation, static aspects are prominent and the limits of informal methods do not affect the quality of the results. However, dynamic aspects quickly become crucial in the software development process. The possibility of simulating and analyzing dynamic aspects already during requirements specification can deeply affect the costs of development and quality of results [3]. Unfortunately informal methods provide weak support to simulation and analysis and thus offer insufficient simulation and analysis capabilities.

UML perfectly mirrors the current state-of-practice: It is syntactically rich, user friendly, simple, and flexible, but lacks the formality required to strongly support simulation and analysis. Scientists ([5]), companies ([10]), and standard organizations ([15]) are trying to overcome the lack of formality in different ways. The approaches investigated so far are mostly limited to static semantic aspects, important, but not sufficient to provide full simulation and analysis capabilities.

This paper indicates a way to complement UML with high-level timed Petri nets (HLTPNs, [8]) to provide dynamic semantics. This paper first overviews various semantic aspects of UML by referring to a simple example. Then, it presents a novel approach to complement UML with HLTPNs, which introduces dynamic semantics without affecting flexibility, user friendliness, and simplicity. Finally, it exemplifies the possibilities of early simulation and analysis supported by the approach.

2   On Formalizing UML

The complete formalization of a notation requires the definition of its concrete and abstract syntax and its static and dynamic semantics. The concrete syntax describes the graphic appearance of notation elements. The abstract syntax identifies the conceptual elements belonging to the notation and indicates the possible relations among them. The static semantics describes the static meaning of the elements and constrains their mutual relations. Finally, the dynamic semantics defines the behaviors described by the notations.

UML is a complex model, which comprises several complementary notations: use cases, class diagrams, interaction diagrams, statecharts, and activity

[1] E-mail: `baresi@elet.polimi.it`
[2] E-mail: `pezze@disco.unimib.it`

diagrams for requirements elicitation and specification. Some of them, such as use cases, mainly describe the structure of the system, while others, e.g., statecharts, focus on the behavior. UML provides a rigorous definition of concrete and abstract syntax for all included notations, but does not fully formalize static and dynamic semantics.

The problem of providing formal semantics to UML has been tackled in different ways. Some work pairs UML with general-purpose formal methods. For example, VDMTools, produced by IFAD, pair the UML class diagram with VDM++, an object-oriented extension to VDM [10]. Other work formalizes specific aspects of UML. For example, France et al. formalize elements of the UML meta-model using Z [5]; Lilius and Paltor formalize UML statecharts to allow UML (specific) models to be analyzable by means of model-checking [14]; Engels et al. ([4]) provide ways for transforming UML interaction diagrams in Java code: The formalization remains implicit and is mandatory to define automatic translation mechanisms. Other work superimposes further constraints to enforce precision. For example, the *pUML* group presents an architectural reorganization of UML based on a sound and rigorous meta modeling language (MML [6]). The outcome has been used to issue a proposal for a more precise UML2.0 ([15]) to the Object Management Group (OMG). They increase the degree of precision of UML by means of well-formedness rules, but do not address dynamic semantics. The target is more on the extensions to UML (profiles) than on its semantics. As another example, we want to mention *Alloy* [12], which does not constrain UML, but proposes similar, but lighter and formally sound notations for specifying object-oriented systems.

For some of the notations that form UML models, in particular the ones that describe the structure of the system such as the use cases or class diagrams, the formal definition of the dynamic semantics is not strictly necessary for defining and analyzing these notations themselves. For other notations, in particular the ones that define the behavior of the system such as statecharts or interaction diagrams, the formal definition of the dynamic semantics is required to provide the needed simulation and execution capabilities. However, to provide a complete dynamic model we need to take into account not only the notations that define behavioral aspects, but also some of the notations that define the structure, to provide a coherent semantic framework for the different views of the system.

In this paper, we illustrate the approach by focusing on class diagrams and statecharts. Class diagrams mainly define the structure of the system. For example, the class diagram shown in Figure 1 indicates that our *Gas station problem* is composed of a *Gas Station*, two *Pumps* and three *Drivers*. It also indicates the main methods of the system elements and the possible interactions among classes. But it does not provide information about the operational behavior of the system, which can be defined through statecharts that specify the operational behavior of classes. For example, the statecharts of Figure 2 describe the states and transitions of the classes of the system.
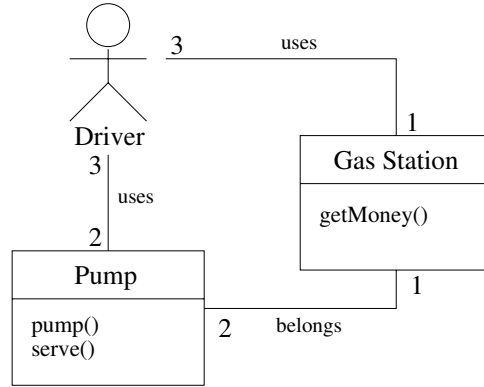
Fig. 1. The Gas Station class diagram

In these diagrams, we followed the convention that events and actions must be formulated in terms of available methods. Thus, a class (statecharts) can listen to events either asking for its methods (*req(<method>)*), or signaling the completion of required methods (*com(<method>)*). As to actions, statecharts can either require an external service (*req(<method>)*) or execute a method (*ser(<method>)*). For the sake of simplicity, we did not consider attributes, which could have produced significant conditions to be evaluated before serving an event.

Although the operational behavior is captured mainly by the statecharts, a formalization of the dynamic semantics requires information provided also in the class diagram (and maybe interaction diagrams) to correctly merge the semantics of the different statecharts. Class diagrams provide the framework, that is, how many classes along with their interfaces. Dynamic diagrams describe how the interfaces are "implemented" and messages are actually exchanged among classes.

3   Adding Dynamic Semantics to UML

The few approaches, proposed so far, that address the problem of providing dynamic semantics to UML follow a traditional schema that provide a fixed mapping to a formal model, thus reducing the flexibility of the informal notation. Such a schema has been widely exploited in the nineties for structural analysis but failed in providing really usable solutions [7]. Although the reasons for failing may be tracked to several causes, our experience within some industrial projects indicates that the reduced flexibility falls among the main causes. A different schema consists in defining a mapping through a set of flexible rules that can easily be extended to cope with different interpretations for the same model. Such a schema has been initially investigated by Paige ([16]) and Baresi et al. ([2]).

In this paper, we suggest an operational schema based on Petri nets for giving formal semantics to UML and we show that a rule-based schema can be adopted to provide flexible semantics to UML. UML presents specific fea-

(a) class Driver

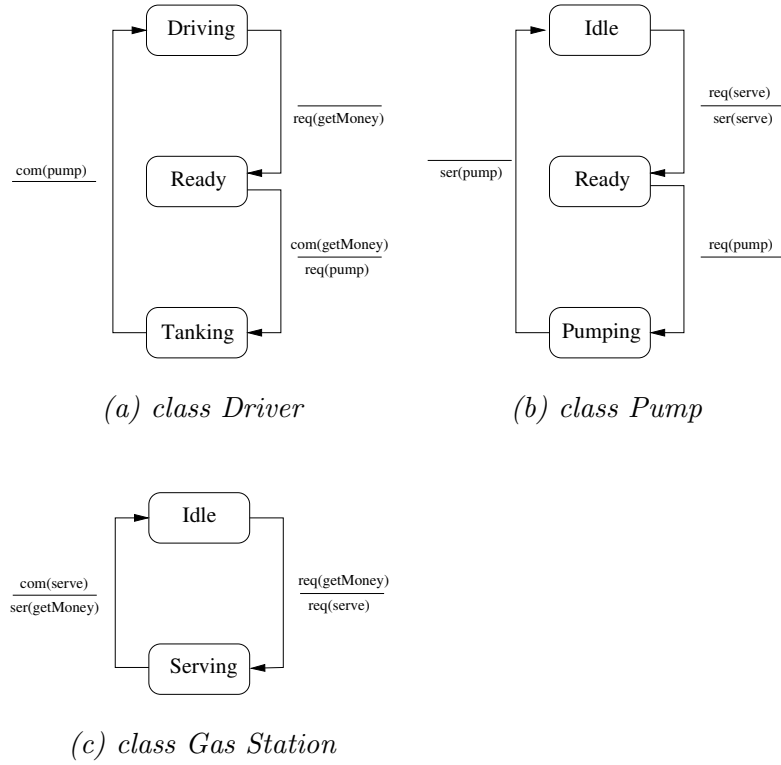(b) class Pump

(c) class Gas Station

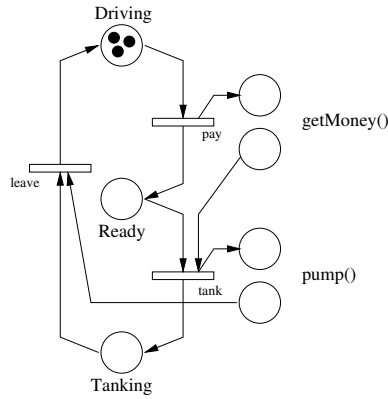Fig. 2. The Statecharts for the Gas Station example

tures that make the extension non trivial: The main ones are the variety of notations used for specifying a system and the absence of a rigid hierarchical framework that constrains a specification. Here we demonstrate the suitability of Petri nets for addressing such problems and the applicability of a rule based schema to UML. We show how to provide formal semantics to a specification composed of a class diagram and some statecharts diagrams, thus we address the problem of providing formal semantics to a heterogeneous set of notations. We also illustrate how the sets of rules for defining the semantics of the different notations can be applied in different orders to address the problem of formalizing a model that lacks a rigid development framework.

Figure 3 shows the Petri net semantics for the statecharts of the Gas Station presented in Section 2. The complete dynamic semantics is given with high-level timed Petri nets, which augment tokens with data, and complement transitions with predicates, actions and time functions[3]. The nets of Figure 3 can be obtained automatically:

- Each class is rendered with as many pairs of places as its methods. Given a method M, a token in place Min means that M has been invoked; a token in $Mout$ means that M has completed its execution.

---

[3] All details of the nets of Figure 3 are presented in the Appendix.

111

- States of the statecharts are modeled with places of the Petri nets, and transitions of the statecharts are modeled with Petri net transitions.

- Relation among states and transitions in the statecharts are modeled with arcs between corresponding places and transitions of the Petri nets.

- Events and actions in the statecharts correspond to requests for services, completions of services and acknowledgements to service completions. The pairs that correspond to provided services have already been added when translating classes; the pairs for required services must be added. The convention used in this paper is that provided services are on the left-hand side of the Petri net, while required services are on the right-hand side.

- Instances are modeled with tokens; thus we present the system with three drivers, a gas station and two pumps.

(a) class Driver   *(b) class Pump*
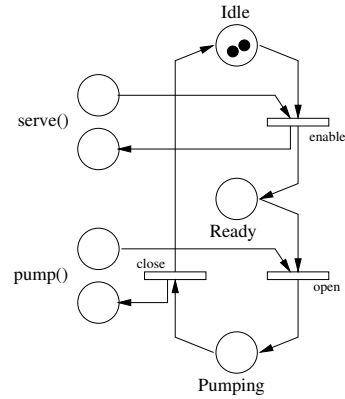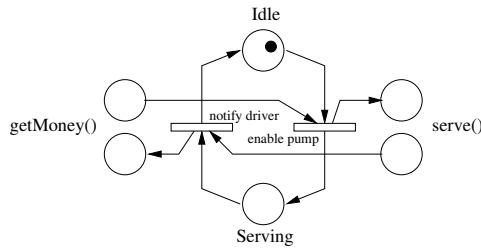
(c) class Gas Station

Fig. 3. The Petri nets for the Gas Station example

- Pair of places are merged by following the rule that two pairs for the same service, one asking for the service and the other supplying it, become a single pair which maintains all previous connections (arcs).

  The way the statecharts (Petri nets) must be connected is not shown explicitly by these diagrams. The feasibility of the connection is part of the class

diagram, but interaction diagrams, which take into account possible dynamic bindings between invocations and actual services, should state the actual connections. In this example, the connections are extremely easy: We can simply collapse the pairs of places with the same names. For example, both the Petri net that corresponds to class *Driver* and the one associated with class *Gas Station* have a pair of places with lable *getMoney*. The driver asks for the service and waits for its completion, the gas station provides the service itself. Thus, the two Petri nets can be connected by collapsing the two pairs of places, leaving incoming and outgoing arcs untouched. This merge operation can be formalized by the rule of Figure 4.

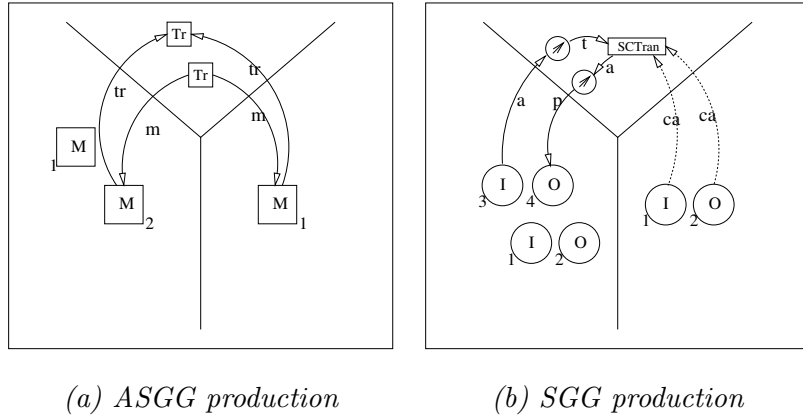(a) ASGG production   (b) SGG production

Fig. 4. Example Rule to Connect Requests to Services

The rule comprises an *ASGG* (Abstract Syntax Graph Grammar) production and an *SGG* (Semantic Graph Grammar) production. The ASGG production defines how the abstract view of the UML model is modified; the SGG production specifies how the functionally equivalent Petri net is modified correspondingly. The ASGG production selects two *method* objects (the two nodes M-labeled on the left-hand side of the production), deletes one of them (node number 2, which appears on the left-hand side, but not on the right-hand side of the production), and connects all *state transition* elements connected to the deleted one with the left one (right-hand side and context). Similarly, the SGG production identifies the two pairs of Petri net places, which correspond to the two *method* objects, deletes the pair corresponding to the deleted node of the syntactic production, and connects the left pair with all transitions the deleted pair was connected to by means of new Petri net arcs. To add a variable number of elements, in this case *arcs*, we use special edges, called meta-edges, which trigger suitable sub-productions. The rule based approach and the use of meta-edges are detailed in [1].

Figure 5 shows the complete Petri net, which corresponds to the whole Gas Station problem, obtained by applying the rule of Figure 4 three times to pair all service requests with the services themselves.
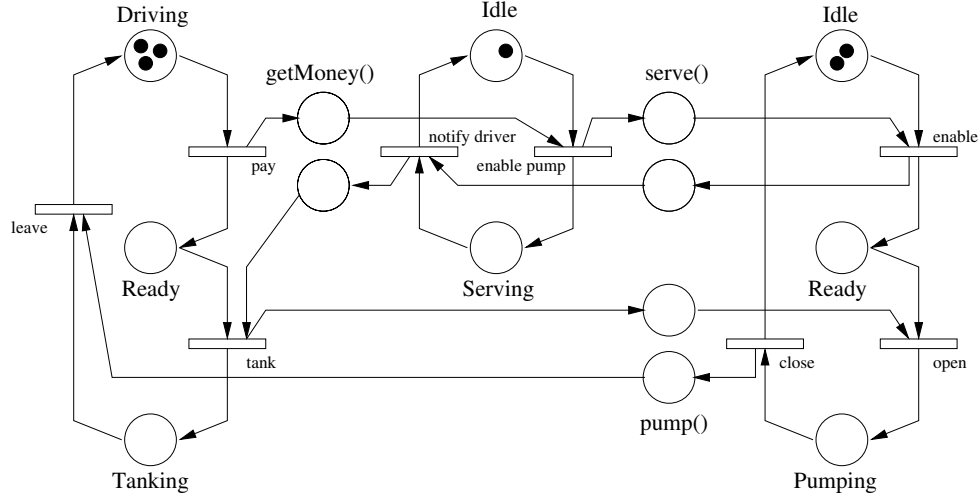
Fig. 5. The complete Petri net

4   Analyzing the UML Semantic Model

By formalizing dynamic semantics aspects of UML, Petri nets allow for formal analysis of UML specifications. Petri nets support various kinds of analyses with different costs and precision. Here we focus on the main techniques for analyzing dynamic aspects: simulation, reachability analysis, and model checking.

Analysis methods work on Petri nets, while software engineers would like to express properties and examine results in terms of their UML specifications. Thus, the schema for mapping UML to Petri nets presented in this paper should be augmented to allow the mapping of results from Petri nets back to UML specifications. Here, we assume that such a mapping exists. Interested readers can refer to [1].

Simulation consists in building firing sequences and visualizing them in terms of UML states. Simulation is computationally inexpensive, fully automatic, and intuitive. It can reveal failures and provide important feedback to end-users, thus supporting both verification and validation of the specifications. For example, the simulation of the Petri net that models the dynamic semantics of the Gas Station problem can help end-users understand the final behavior of the system, and thus early validate the specification, and can reveal possible failures. The execution sequence illustrated in Figure 6, which can be obtained by simulating the Petri net, is incorrect since it allows drivers to pump different amounts of gas with respect to what they pay for.

In this case, the failure can be tracked back to a design error: the software engineer did not identify the drivers enabled to pump from ready pumps, thus *Driver1* can pump from *Pump2*, enabled for *Driver2*. Unfortunately the low computational cost of simulation is paid in terms of analysis power. Simulation can reveal failures anymore, but cannot prove absence of undesirable behaviors. Thus, once corrected the fault in the specification, simulation will

Petri net Firings	D1	D2	D3	GS	P1	P2
	Driving	Driving	Driving	Idle	Idle	Idle
pay(D1)	Ready	Driving	Driving	Idle	Idle	Idle
enablePump(P1)	Ready	Driving	Driving	Serving	Idle	Idle
enable(P1)	Ready	Driving	Driving	Serving	Ready	Idle
notifyDriver(D1)	Ready	Driving	Driving	Idle	Ready	Idle
pay(D2)	Ready	Ready	Driving	Idle	Ready	Idle
enablePump(P2)	Ready	Ready	Driving	Serving	Ready	Idle
enable(P2)	Ready	Ready	Driving	Serving	Ready	Ready
notifyDriver(D2)	Ready	Ready	Driving	Idle	Ready	Ready
tank(D1)	Tanking	Ready	Driving	Idle	Ready	Ready
open(P2)	Tanking	Ready	Driving	Idle	Ready	Pumping
close(P2)	Tanking	Ready	Driving	Idle	Ready	Idle
leave(D1)	Driving	Ready	Driving	Idle	Ready	Idle
tank(D2)	Driving	Tanking	Driving	Idle	Ready	Idle
open(P1)	Driving	Tanking	Driving	Idle	Pumping	Idle
close(P1)	Driving	Tanking	Driving	Idle	Idle	Idle
leave(D2)	Driving	Driving	Driving	Idle	Idle	Idle

Petri net firings are indicated with a transition name and a token identifier. The token identifier indicates the token that causes the transition to fire either among the three drivers (identifier Di) or among the two pumps (identifier Pi). Columns D1, D2, D3, GS, P1, and P2 correspond to the states of the three statecharts of type Driver (Di), the statechart Gas Station (GS), and the two statecharts of type Pump (Pi). They indicate the evolutions of the states of the statecharts modeled by the Petri net firing sequence in the leftmost column. Each tuple corresponds to the states entered by the statecharts after the firing of the transition on the same line.

Fig. 6. A firing sequence of the Petri net semantics of the Gas Station that reveals a failure in the UML specification

not reveal the failure, but we cannot conclude that the specification is fault free.

In contrast, reachability analysis and model checking can prove the validity of particular properties, and thus the absence of certain classes of faults, albeit at higher computational costs. Reachability analysis consists of finite

enumeration of all possible states of the computation. Such a technique has been proved feasible for Place/Transition nets, regardless of the boundedness of the markings, and has been extended to various kind of high-level Petri nets ([13]) including the high-level timed Petri nets used in this paper to give dynamic semantics to UML [9]. Model checking applied to a reachability graph can be used to prove several interesting liveness and safety properties. For example, model checking can prove that a UML specification of the Gas Station problem always sells the due amount of gas to the customers, or that it never allows a customer to get gas without paying for it.

The formal definition of the dynamic semantics of UML specifications can thus anticipate important checks, reduce the risks of propagating faults from early specifications to final code, and decrease the costs related to fault identification and removal.

5 Conclusions

The paper briefly discusses the main problems with ascribing formal semantics to the dynamic models of UML. It exemplifies a rule-based pairing of UML with high-level timed Petri nets through a Gas Station problem. Besides sketching the way to derive the Petri nets from UML diagrams, we propose also some ways to exploit the Petri nets to simulate and analyze UML models already when defining the system's requirements.

The approach is only sketched and is still the target of our research. In the near future, we will concentrate on:

- Completing the set of rules needed to address all significant UML diagrams. Currently, we are addressing only the main elements.

- Stressing simulation and analysis capabilities to get as much significant information as possible from the Petri nets. We feel that this information is extremely important and useful to designers.

- Stressing the integration with available CASE tools to supply users with an integrated environment.

- Exploiting the same approach to "formalize" UML-RT and maybe other significant extensions.

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] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[4] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and Their Transformation to Java. In *Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard.* Volume 1723 of *Lecture Notes in Computer Science*, pages 473–488. Springer-Verlag, 1999.

[5] A. Evans, R. France, K. Lano, and B. Rumpe. Developing the UML as a Formal Modelling Notation. In *Proceedings of UML'98 - The Unified Modeling Language. Beyond the Notation.* Volume 1618 in *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[6] A. Evans and S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In *Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard.* Volume 1723 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1999.

[7] R. France and M. Larrondo-Petrie. From Structured Analysis to Formal Specifications: State of the Theory. In *Proceedings of Computer Science Conference*, pages 249–256. ACM Press, April 1994.

[8] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A Unified High-Level Petri Net Model For Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.

[9] C. Ghezzi, S. Morasca, and M. Pezzè. Validating Timing Requirements for Time Basic Net Specifications. *The Journal of Systems and Software*, 27(2):97–109, November 1994.

[10] The VDM Tool Group. The IFAD VDM++ Language. Technical report, IFAD, February 2000.

[11] C. Heitmeyer. On the Need for "Practical" Formal Methods. Volume 1486 of *Lecture Notes in Computer Science*, pages 18–30. Springer-Verlag, 1998.

[12] D. Jackson. Alloy: A Lightweight Object Modelling Notation. Technical Report, MIT, July 2000.

[13] K. Jensen. *Coloured Petri Nets*. Springer-Verlag, Berlin, 1996.

[14] 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-CS, 1999.

[15] OMG. UML 2.0 RFI. Technical report, OMG, August 1999.

[16] R.F. Paige. A Meta-Method for Formal Method Integration. Volume 1313 of *Lecture Notes in Computer Science*, pages 473–485, 1997.

[17] H. Saiedian. An Invitation to Formal Methods. *IEEE Computer*, pages 16–30, April 1996.

A   HLTPN Details

In this section we use a Java-like syntax to describe the textual annotations associated with the nets of Figure 3.

The following type definitions clarify the information associated with each token:

```
class Driver {
  int id;
  String name;
}

class Pump {
  int id;
  int drv;
}

class GasStation {
  int drv;
}
```

Places *Driver*, *Ready*, and *Tanking* of Figure 3(a) contain tokens of class `Driver`. Places *Idle*, *Ready*, and *Pumping* if Figure 3(b) contain tokens of class `Pump`. Places *Idle* and *Serving* of Figure 3(c) contain tokens of class `GasStation`.

These definitions allow us to specify also predicates and actions associated with each transition. For example, transition *pay* of Figure 3(a) can be specified as follow:

```
Transition pay
enab: 0,0;
predicate: true;
action: getMoneyOut = Driving.id;
        Ready = Driving;
```

It must fire immediatly (its enabling interval is 0, 0) after having a token in each place of its preset (the predicate is true). It produces a token to ask for the service and moves the driver from driving to ready.

Similarly, transition *close* of Figure 3(b):

```
Transition close
enab: t1,t2;
predicate: true;
action: Idle = Pumping;
        pumpOut = Pumping.drv;
```

This means that the transition firing must be delayed by from t_1 to t_2 time units. As soon as there is a token in place *Pumping*, the transition is enabled and we can start couting the delay. Its firing "copies" the token in *Pumping* to *Idle* and produces an token in place *pumpOut* to signal driver who releases the pump.

The specification of all other transitions does not need futher comments:

```
Transition tank
```

```
        enab: 0,0;
        predicate: getMoneyIn = Ready.id;
        action: Tanking = Ready;
                pumpOut = Ready.id


Transition leave
enab: 0,0;
predicate: pumpIn = Tanking.id;
action: Driving = Tanking;


Transition enable
enab: 0,0;
predicate: true;
action: Ready = Idle;


Transition open
enab: 0,0;
predicate: true;
action: Pumping = Ready;
        Pumping.drv = pumpIn;


Transition enablePump
enab: 0,0;
predicate: true;
action: Serving.drv = getMoneyIn;


Transition notifyDriver
enab: 0,0;
predicate: true;
action: getMoneyOut = Serving.drv;
```