Specification of UML Classes by Object Oriented Petri Nets

Radek Kočí and Vladimír Janoušek

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
{*koci,janousek*}*@fit.vutbr.cz*

***Abstract*—The UML class diagram defines a basic architectonic model of the system. Its behavior is then usually described by other UML diagrams, such as activity diagrams, sequence diagrams, etc. These models serve for the design purposes and are automatically or manually transformed in the next development stages, typically to the models with formal basis or to implementation (production) environment. There is no backward step allowing to investigate the system structure and its behavior with the designed models. On the other hand, there are approaches to system design combining design, testing, and implementing stages into one development technique. One of them uses Object Oriented Petri Nets (OOPN) as basic modeling formalism. Nevertheless, OOPN lacks for advisable architectonic view of modeled systems as it is offered by UML class diagram. The paper is aimed at using UML class diagrams for system architecture description and the OOPN formalism for description of classes behavior. Since UML classes and OOPN classes partially differs, we define formal transformation between UML classes and OOPN classes.**

***Keywords*-Class diagram; Object-Oriented Petri Nets; UML; transformation.**

I. Introduction

Design methodologies use models for system specification, i.e., for defining the structure and behavior of developed system. The most popular modeling language in software engineering is UML [1]. It serves as a standard for analytics, designers and programmers. But, own phraseology of UML does not have enough power allowing to realize some fundamental relationships and, in particular, rules, that are branch of every modeled system. To model dynamic aspects of the system, the designer usually describes them by static diagrams in a design phase and he cannot make certain of his partial ideas about the system behavior. Although the UML language can be completed by extensions, e.g., OCL (Object Constraint Language), making the system description more precise, it makes the checking of models correctness or validity by means of simulation complicated.

Therefore, new methodologies and approaches are investigated and developed for many years. They are commonly known as Model-Driven Software Development or Model-Based Design (MBD) [2], [3], [4]. An important feature of these methods is the fact that they use executable models, e.g., Model Driven Architecture (MDA) [5] and Executable UML [6], allowing to simulate models, i.e., to provide simulation testing. The created models can be (semi)automatically transformed to implementation language (the code generation). Nevertheless, the result has to be finalized manually, so it entails a possibility of semantic mistakes or imprecision between models and transformed code.

There are other similar methods that use the pure formal models (e.g., Petri Nets, calculus, etc.) allowing to use formal or simulation approaches to complete the design, testing, and implementation activities. In comparison with semi-formal models, formal models bring clear and understandable modeling and the possibility to test correctness with no need for model transformations. The design method, which is taken into account in this paper [7], [8], derives benefit from formalisms of Object Oriented Petri Nets (OOPN) [9], [10]. The paper is aimed at the class description using Object Oriented Petri Nets (OOPN). Since the UML classes and OOPN classes partially differ, we define formal transformation between UML classes and OOPN classes and formal constraints the classes and objects have to satisfy. The goal is to keep an eye to the system at the architectonic view with UML and at the behavioral view with the formalism of OOPN.

The paper is organized as follows. First, we briefly introduce used design methodology in Section III. Then the formalisms will be described in Section IV. Section V introduces relationships between UML classes and OOPN classes and a mechanism of class transformations. The proposed mechanism will be demonstrated with the example in Section VI.

II. Related Work

The are works that are similar to the proposed one. First, the formalism of nets-within-nets (NwN) was introduced by Valk [11] and Moldt [12], [13]. The formalism of NwN is similar to OOPN, but OOPN fully support an integration of formal description of objects and objects from target environment, which facilitates, e.g., reality-in-the-loop simulation or usage of formal models into target application. Second, there are tools merging UML and Petri nets, for instance ArgoUML [14]. The difference is similar to the previous situation—these tools allow to *model* systems using

 ISBN: 978-1-61208-230-1

combination of different formalisms, but do not allow to use formal models in system *implementation*.

III. Design Methodology

The design methodology [15] stems from the classic approach of class identification and definition and extends it to the new features. Primarily, there have to be found essential objects of a modeled system and their relationships. There we can successfully employ resources of UML such us *Use Case*, *Activity*, and *Class* diagrams. Thus, the design process comprises, among others, the identification of *use cases* of the system and the specification of *classes* and their behavior. To specify the behavior, the methodology distinguishes *roles* and *activity nets* as a special kinds of classes. These mentioned nets represent appropriate roles and use cases in the system and are layered hierarchically. Each role encapsulate activity nets and, moreover, each role can encapsulate another role. It allows to get a new view to the role based on the existing one.

Each role has its own set of allowed activities (activity nets) described by OOPN. If anybody wants to perform the activity, it has to ask the role for creating an instance of the activity and then it can use this activity as a use case of the system. The execution of nets are synchronized by means of synchronous ports. The nested nets define synchronous port for synchronization of executions and the net at higher layer is controlled by calling these ports. This principle will be demonstrated at the appropriate places in following parts.

IV. Formalisms

We will present a short introduction to formalisms and models used in this section.

A. Structural and Behavioral Views with UML

The UML modeling [1] uses a notion of *view*. A view of a system is a projection of the system on one of its relevant aspects. Such a projection focuses on certain aspects and ignores others. For our purposes, we mention only two views. The *structural view* describes layout between objects and classes, their associations and their possible communication channels. As an example, we can mention *Class diagram*. The *behavioral view* describes, how the system components interact, and characterizes the response to external system operations. For our purposes, we will not use UML diagrams, but OOPN for behavioral view.

B. Formalism of OOPN

An *Object Oriented Petri net* (OOPN) is a set of classes specified by high-level Petri nets. Formally, OOPN comprises constants $CONST$, variables VAR, net elements (such as places P and transitions T), class elements (such as object nets $ONET$, method nets $MNET$, synchronous ports $SYNC$, negative predicates $NPRED$ and message selectors MSG), classes $CLASS$, object identifiers OID, and method net instance identifiers MID. We denote $NET = ONET \cup MNET$ and $ID = OID \cup MID$.

A *class* is mainly specified by an object net (an element of $ONET$), a set of synchronous ports and negative predicates (a subset of $SYNC$ and $NPRED$), a set of method nets (a subset of $MNET$), and a set of message selectors (a subset of MSG) corresponding to its method nets, synchronous ports, and negative predicates. Object nets describe possible autonomous activities of objects, while method nets describe reactions of objects to messages sent to them from the outside.

An example illustrating the important elements of the OOPN formalism is shown in Figure 1. There are depicted two classes `C0` and `C1`. The object net of the class `C0` consists of places `p1` and `p2` and one transition `t1`. The object net of the class `C1` is empty. The class `C0` has a method `init:`, a synchronous port `get:`, and a negative predicate `empty`. The class `C1` has a method `doFor:`.

Synchronous ports are special (virtual) transitions, which cannot fire alone but only dynamically fused to some other transitions, which activate them from their guards via message sending. Every synchronous port embodies a set of conditions, preconditions, and postconditions over places of the appropriate object net, and further a guard, and a set of parameters. Parameters of an activated port s can be bound to constants or unified with variables defined on the level of the transition or port that activated the port s. An example is shown in Figure 1, the port named `get:` having one parameter `o`. This port is called from the transition `t2` (class `C1`) with unbound variable `n`—it means that the variable `n` will be unified with the content of the place `p2` (class `C0`).

Negative predicates are special variants of synchronous ports. Its semantics is inverted—the calling transition is fireable if the negative predicate is not fireable. The passed variable cannot be unbound (the unification is impossible) and the predicate cannot have a side effect. An example is shown in Figure 1, the predicate named `empty`. This predicate is called from the transition `t3` (class `C1`)—it means that the transition `t3` will be fireable if the place `p2` (class `C0`) will be empty.

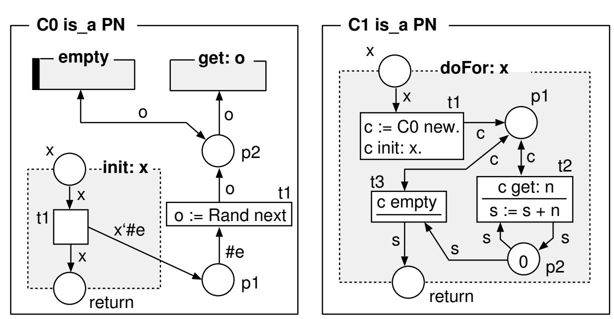

Figure 1. An OOPN example.

V. RELATIONSHIP BETWEEN UML AND OOPN CLASSES

We will present a relationship between classes of UML and OOPN and their reciprocal mapping.

A. Prerequisites

First, we define formal structures that will be used in next definitions. In pure object systems, everything is understood as an object, so that there is no requirement for defining special kind of types. Nevertheless, for our purpose we define $TYPE = CLASS \cup OCLASS \cup \{\varepsilon\}$, where $CLASS$ is a set of domain (OOPN) classes, $OCLASS$ is a set of other types (e.g., classes from the production environment or primitive types), and ε represents a special kind of type meaning *unspecified type*. Let the symbol $\triangleleft$ determines a relationship *is of a type* (*is an instance of*). For example, $o \triangleleft A$ means that the object (value) referred by the variable o is an instance of a class A (is of a type A).

The class can be defined as a tuple (n, V_C, I_C, B_C), where n is a class name, V_C is a set of instance variables, I_C is an interface (a set of operations), and B_C is a behavior, usually defined as a set of methods. The OOPN class be alternatively defined as a tuple $(n, P_{ON}, I_{PN}, B_{PN})$, where n is a class name, P_{ON} is a set of places from the object net representing instance variables, $I_{PN} \subseteq MSG$ is an interface, and B_{PN} is a behavior.

B. Interface

The interface of an OOPN class is defined as a subset of message selectors $I_{PN} \subseteq MSG$, where $MSG = MSG_M \cup MSG_S \cup MSG_P$. MSG_M corresponds with method nets, MSG_S corresponds with synchronous ports, and MSG_P corresponds with negative predicates.

There are several ways how I_C can be mapped to I_{PN}. Let f_I be a non-specific mapping $I_C \rightarrow MSG_M$. In this case, each operation is mapped into a message selector of a method net. This way is easy, but not sufficient for design methods that use Petri Nets [15]. Therefore, the operations from I_C are classified into three groups: *action group* $I_C^{Act} \subseteq I_C$ performing some actions on the object; *test group* $I_C^T \subseteq I_C$ performing some tests on the object, and *access group* $I_C^{Acc} \subseteq I_C$ which sets or gets a value of an instance variable. Analogically, let us define I_{PN}^{Act}, I_{PN}^{Acc}, and I_{PN}^T for the OOPN class. Then, the second way of mapping defines specific functions for appropriate group:

$$f_I^{Act} : I_C^{Act} \rightarrow I_{PN}^{Act}, \text{where } I_{PN}^{Act} = MSG_M \cup MSG_S$$
$$f_I^{Acc} : I_C^{Acc} \rightarrow I_{PN}^{Acc}, \text{where } I_{PN}^{Acc} = MSG_M \cup MSG_S$$
$$f_I^{T} : I_C^{T} \rightarrow I_{PN}^{T}, \text{where } I_{PN}^{T} = MSG_S \cup MSG_P$$

The action and access groups are mapped into the same subset of selectors of method nets and synchronous ports. The synchronous ports can influence on the object net during its firing (e.g., an object can be removed from or put into places in an object net), so that the calling a synchronous port from the interface has a direct effect in changing an object net state. Consequently, it can cause an activity of an object net. The negative predicate cannot have any side effects from the definition, so it cannot be a part of action and access groups. The testing group is mapped into a subset of synchronous ports or negative predicates—it depends on the positive or negative sense of the testing.

We can suppose, that the following statement holds for the UML class: $I_C^{Act} \cap I_C^T \cap I_C^{Acc} = \emptyset$. It means, that each operation is a member of only one group. For OOPN class, we can say $I_{PN}^{Act} \cap (I_{PN}^{Acc} \cup I_{PN}^T) = \emptyset$. It means, that operations from I_{PN}^{Act} cannot be members of other groups. Due to the definition of synchronous ports, the same synchronous port can serve for testing as well as for data accessing, so $I_{PN}^{Acc} \cap I_{PN}^T$ not have to be $\emptyset$.

C. Instance variables and types

A mapping of instance variables is defined as an injection $f_V : V_C \rightarrow P_{ON}$, where P_{ON} is a set of places of the object net. The consequence is that the variable is always a multiset of values. If the only one value has to be assigned to the place, as for an ordinary variable, it is possible to define a constraint, see Section V-D. The place in OOPN has assigned no type. But, for analysis and testing purpose, it is possible to 1) assign a set of types the objects can be of, 2) derive a set of types the objects are of from the model analysis or simulation.

Let T_P be a surjection $T_P : P \rightarrow \mathcal{P}(TYPE)$ assigning a set of types to a given place. The type of the place can be derived from the associations between classes, whereas there is no necessary to define only one type (and, thus, to allow all subtypes), but the set can be extended to next types. Implicitly, each place has assigned a type ε.

D. Constraints

Although the OOPN classes bring more intuitive modeling of behavior, they do not offer intrinsic definitions of invariants, a state of the place, or type checking. Nevertheless, there is very simple way how to define and test these conditions by means of OOPN. The advantage of this approach is that the designer has this feature under the control. We will call these definitions as *constraints*. Each such a constraint is defined formally and the definition is followed by its implementation in OOPN showed in Figure 2.

The test of *empty place* is defined as $\varphi(p) = \nexists x \in p$. It is implemented by the negative predicate `emptyPlace` in the OOPN formalism (see Figure 2). If there is no object in the place, the condition is not satisfied and it implies, that the negative predicate is evaluated as true.

The test of *nonempty place* is defined as $\psi(p) = \exists x \in p$. It is implemented by the synchronous port `nonEmptyPlace` in the OOPN formalism (see Figure 2). If there is at least

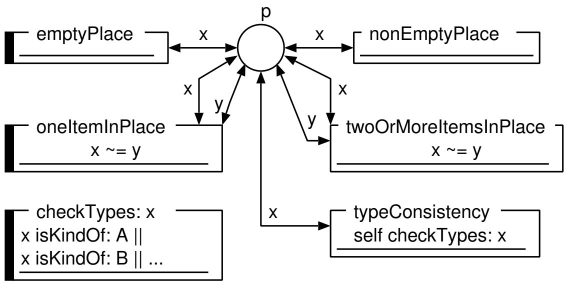

Figure 2. Invariants and testing conditions.

one object in the place, the condition is satisfied and the synchronous port is evaluated as true.

The test of *at most one item* (or *the capacity of the place is 1*) is defined as $\tau(p) = \nexists x, y \in p : x \neq y$. It is implemented by the negative predicate `oneItemInPlace` in the OOPN formalism (see Figure 2). If there is no object or only one object in the place, the conditions are not satisfied and the negative predicate is evaluated as true. In the other cases, it is evaluated as false.

The test of *two or more items* is defined as $\varsigma(p) = \exists x, y \in p : x \neq y$. It is implemented by the synchronous port `twoOrMoreItemsInPlace` in the OOPN formalism (see Figure 2). If there are at least two different objects in the place p, the synchronous port is evaluated as true.

The test of *type consistency* is defined as $\theta(p, ET) = \psi(p) \wedge \exists x \in p : \nexists t \in ET \wedge x \lhd t$. It is implemented by the synchronous port `typeConsistency` and the associated negative predicate `checkTypes:` in the OOPN formalism (see Figure 2). If there is an object x in the place p and there is no type t from the expected types set ET, the conditions of the negative predicate are not satisfied and it implies the negative predicate is evaluated as true. Then the synchronous port is evaluated as true for the object x—it means that this object x does not satisfy the expected types of the place p.

E. Behavior

The behavior B_{PN} is not simply a set of methods because the synchronous ports from interface can influence on the object net during its firing, as mentioned in Section V-B. The object net $n \in ONET$ is defined as a graph of Petri nets. The concrete behavior is usually provided by its part—a valid subnet of the Petri net graph. So we can define $\mathcal{S}(ONET)$ as a set of all valid subnets of the object nets. Then, the behavior B_{PN} can be defined as $B_{PN} \subseteq MNET \cup \mathcal{S}(ONET)$.

VI. Example

This section will present the relationship between UML and OOPN classes. To demonstrate this relationship, a very small part of the PNtalk system [16] was chosen. PNtalk is the tool intended to model and to simulate systems using OOPN. We depict a functionality of the method look-up.

A. UML Class Diagram

By following the design methodology [7], [15], we have to identify roles and use cases and classify them into classes. In the example, the only one role of *object* is identified and its use case *lookFor* (it does not strictly correspond with the real system, but for demonstration it is sufficient). These elements are classified into two classes, the class `Object` for the role and `LookFor` for the activity of method searching (the use case).

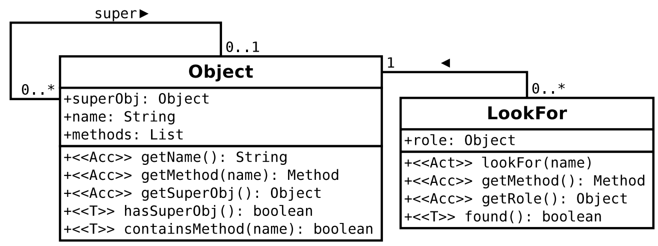

Figure 3. The class diagram of the method look-up.

The `Object` has attributes of the object name, the object's superobject (in the terms of inheritance hierarchy), and the list of object's methods. It offers methods for getting values of attributes (see the stereotype `<<Acc>>` in Figure 3) and methods for testing the object's state (see the stereotype `<<T>>` in Figure 3).

The `LookFor` has an attribute of the role the activity is intended for. It offers a method for the look-up (see the stereotype `<<Act>>` in Figure 3), a method for testing the result of searching (see the stereotype `<<T>>` in Figure 3), and methods for getting values (see the stereotype `<<Acc>>` in Figure 3).

B. The class Object

Let us analyze the class `Object`. It contains three instance variables, so that there will be three places in the OOPN class, according to the function $f_V(Object) = \{name \rightarrow name, methods \rightarrow methods, superObj \rightarrow superObject\}$.

We can identify the following operations from the interface: $I_C^{Act}(Object) = \emptyset$, $I_C^{Acc}(Object) = \{getName, getMethod, getSuperObj\}$, $I_C^{T}(Object) = \{hasSuperObj, containsMethod\}$. The class `Object` offers no operations in I_C^{Acc}, so that there is nothing to transform. There are three operations in $I_C^{Acc}(Object)$, that are transformed into synchronous ports: $f_I^{Acc}(Object) = \{getName \rightarrow name{:}, getMethod \rightarrow method{:}named{:}, getSuperObj \rightarrow superObject{:}\}$.

The test group $I_C^{T}(Object)$ offers two operations, that are transformed into synchronous ports and negative predicates: $f_I^{T}(Object) = \{hasSuperObj \rightarrow \{superObject{:}, notSuperObject\}, containsMethod \rightarrow$

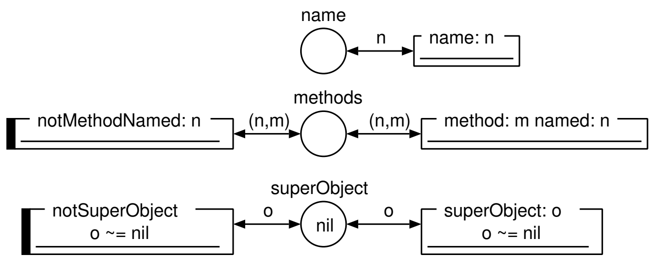

Figure 4. The OOPN class `Object`.

$\{method{:}named{:}, notMethodNamed{:}\}\}$. The synchronous ports allow to get a value of instance variables (using the unification principle) and, at the same time, to test if the variable contains a given value. So, the test operation is transformed usually into a pair of a synchronous port (it allows also for accessing, so that it is a part is the access group) and a negative predicate.

Finally, the interface of the OOPN class `Object` is defined as follows: $I^{Act}_{PN}(Object) = \emptyset$, $I^{Acc}_{PN}(Object) = \{name{:}, method{:}named{:}, superObject{:}\}$, $I^{T}_{PN}(Object) = I^{Acc}_{PN}(Object) \cup \{notMethodNamed{:}, notSuperObject\}$. The graphic notation is shown in Figure 4.

C. The class LookFor

Let us analyze the class `LookFor`. It contains one instance variable, so that there will be one place in the OOPN class, according to the function $f_V(LookFor) = \{role \rightarrow role\}$.

We can identify the following operations from the interface: $I^{Act}_{C}(LookFor) = \{lookFor\}$, $I^{Acc}_{C}(LookFor) = \{getMethod, getRole\}$, $I^{T}_{C}(LookFor) = \{found\}$. There is one operation in $I^{Act}_{C}(LookFor)$, which is transformed into the synchronous port $f^{Act}_{I}(LookFor) = \{lookFor \rightarrow lookFor{:}\}$. There are two operations in $I^{Acc}_{C}(LookFor)$, that are transformed into the synchronous ports $f^{Acc}_{I}(LookFor) = \{getRole \rightarrow role{:}, getMethod \rightarrow found{:}\}$. There is one operation in $I^{T}_{C}(LookFor)$, which is transformed into the synchronous port and the negative predicate $f^{T}_{I}(LookFor) = \{found \rightarrow \{found{:}, failed\}\}$ testing the positive or negative state of the search result.

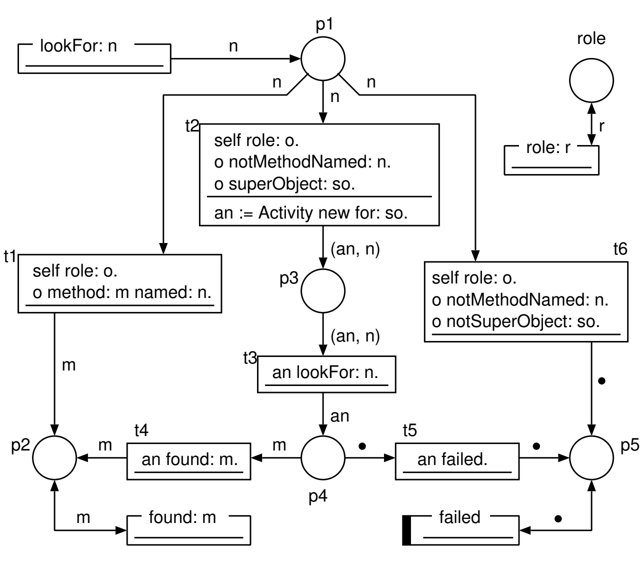

Figure 5. The OOPN class `LookFor`.

Finally, the interface of the OOPN class `LookFor` is defined as follows: $I^{Act}_{PN}(LookFor) = \{lookFor{:}\}$, $I^{Acc}_{PN}(LookFor) = \{role{:}, found{:}\}$, $I^{T}_{PN}(LookFor) = I^{Acc}_{PN}(LookFor) \cup \{failed\}$. The graphic notation is shown in Figure 5.

D. Behavior

The behavior of the activity net `LookFor` can be divided into three basic subnets (the subnet is described as a set of vertexes, i.e., places and transitions): $\delta_1 = \{p1, t1, p2\}$, $\delta_2 = \{p1, t2, p3, t3, p4, t4, p2, t5, p5\}$, and $\delta_3 = \{p1, t6, p5\}$. The δ_1 is a behavior for a situation if the method is found directly in the object (see the transition $t1$). The δ_3 is a behavior for a situation if the method is not found directly in the object and the object does not have an superobject (see the transition $t6$). The δ_2 is a behavior for a situation if the method is not found directly in the object and the object has an superobject. Then the new activity net is created for the superobject (see the transition $t2$). Then the operation $lookFor{:}$ is called (the transition $t3$) and the result is tested (the transitions $t4$ and $t5$). The places $p2$ and $p5$ store the state of the operation, which can be tested by $found{:}$ and $failed$. The synchronous port $found{:}$ serves even as an access operation for getting the found method.

E. Constraints

Now, we demonstrate an usage of constraints in the class definition. We chosen the place $superObject$ from the class `Object`. First, the place is initialized by a special value nil representing an information that the object does not have a superobject. If the object has an superobject, the value nil is replaced. So there is one invariant: *the place $superObject$ contains just one value*. This constraint is tested by $\varsigma(superObject)$. Second, the place can contain only objects of a type `Object`. This constraint is tested by $\theta(superObject, \{Object\})$. Declaration of both constraints in the OOPN class is shown in Figure 6a.

The constraints are realized by synchronous ports or negative predicates. Their definition does not evocate any activity or testing without its calling. Hence, it is possible to define many constraints on the classes with no influence on the system performance. In order to activate the tests, they have to be called, as shown in Figure 6b. The tested object is stored in the place p and the associated transitions provide the appropriate tests. These transitions can be a part

 ISBN: 978-1-61208-230-1

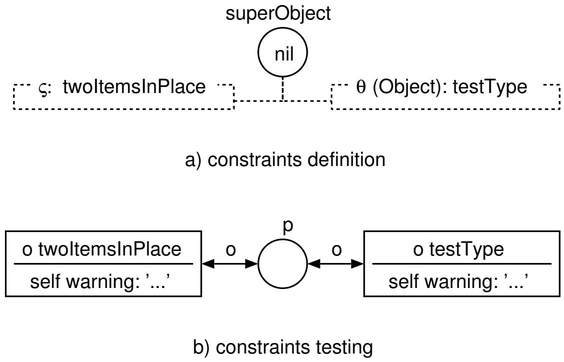

Figure 6. The class `Object`: a) constraints definition and b) testing.

of any object nets (then the transition is fired immediately the condition occurs) or any method net (then the tests are provided on demand).

VII. Conclusion and Future Work

The paper dealt with a formal approach to describe system structure and behavior. Proposed approach extends system modeling using formalism of Object Oriented Petri Nets (OOPN) with selected UML diagrams. First, the class diagram was taken into account. The approach stems from UML classes for system structure specification, where classes behavior is modeled by OOPN. Since the UML classes and OOPN classes differ, the transformation technique has been introduced. The presented approach is a part of the development methodology, which allows to use formal models in all phases of system development. Formal models should be used as basic design, analysis and also programming means with a vision to allow for combining of simulated and real components and to deploy models as the target system with no code generation. Using UML classes together with formalism of OOPN satisfies the development methodology, because one-to-one assignability enables to keep an eye to the system with UML and OOPN formalisms and, together, to use OOPN models as a programming means. In the future, we plan to complete transformation mechanisms with class associations, extend modeling with use case diagrams, and investigate simulation techniques for an assistance in the system modeling.

Acknowledgment: This work has been supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by BUT FIT grant FIT-S-11-1, and by the Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

[1] J. Arlow and I. Neustadt, *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Professional, 2001.

[2] S. Beydeda, M. Book, and V. Gruhn, *Model-Driven Software Development*. Springer-Verlag, 2005.

[3] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[4] M. Broy, J. Gruenbauer, D. Harel, and T. Hoare, Eds., *Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute*. Kluwer Academic Publishers, 2005.

[5] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, ser. 17 (MS-17). John Wiley & Sons, 2003.

[6] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie, *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[7] R. Kočí and V. Janoušek, "System Design with Object Oriented Petri Nets Formalism," in *The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008*. IEEE Computer Society, 2008, pp. 421–426.

[8] R. Kočí and V. Janoušek, "OOPN and DEVS Formalisms for System Specification and Analysis," in *The Fifth International Conference on Software Engineering Advances*. IEEE Computer Society, 2010, pp. 305–310.

[9] M. Češka, V. Janoušek, and T. Vojnar, *PNtalk — a Computerized Tool for Object Oriented Petri Nets Modelling*, ser. Lecture Notes in Computer Science. Springer Verlag, 1997, vol. 1333, pp. 591–610.

[10] R. Kočí and V. Janoušek, *Simulation Based Design of Control Systems Using DEVS and Petri Nets*, ser. Lecture Notes in Computer Science. Springer Verlag, 2009, vol. 5717, pp. 849–856.

[11] R. Valk, "Petri Nets as Token Objects: An Introduction to Elementary Object Nets." in *Jorg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science*, vol. 120. Springer-Verlag, 1998.

[12] D. Moldt, "OOA and Petri Nets for System Specification," in *Object-Oriented Programming and Models of Concurrency*. Italy, 1995.

[13] L. Cabac, M. Duvigneau, D. Moldt, and H. Rölke, "Modeling dynamic architectures using nets-within-nets," in *Applications and Theory of Petri Nets 2005. 26th International Conference, ICATPN 2005, Miami, USA*, 2005, pp. 148–167.

[14] Tigris.org, "ArgoUML: open source UML modeling tool," http://argouml.tigris.org/, 2012.

[15] R. Kočí and V. Janoušek, "Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study," in *Proceeding of the International Workshop on Petri Nets and Software Engineering 2012*, vol. 851. CEUR, 2012, pp. 253–266.

[16] V. Janoušek and R. Kočí, "Embedding Object-Oriented Petri Nets into a DEVS-based Simulation Framework," in *Proceedings of the 16th International Conference on System Science*, ser. volume 1, 2007, pp. 386–395.