On the Object Oriented Petri Nets Model Transformation into Java Programming Language

Radek Kočí

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 2, 612 66 Brno, Czech Republic
email: koci@fit.vut.cz

***Abstract*—Nowadays, high-level languages and approaches to software design and implementation are often used. The main reasons for this are the possibility of faster and more efficient design and more efficient verification of the design produced. To deploy a system created in this way, either a framework can be used to handle the formalism used or the design can be transformed into a programming language or lower-level formalism. In this paper, we focus on the Object Oriented Petri Net (OOPN) formalism and introduce the idea of transforming OOPN models into the Java programming language.**

***Keywords*—Object Oriented Petri Nets; model transformation; Java.**

I. INTRODUCTION

The key activities in system development are specification, testing, validation, and analysis (e.g., performance or throughput). Most methodologies use models for system specification, i.e., to define the structure and behavior of the system under development. There are different kinds of models, ranging from low-level formal-based models to purely formal models. Each kind has its advantages and disadvantages. Less formal models, e.g., Unified Modeling Language (UML), allow the basic concepts of a system to be quickly described; on the other hand, they do not allow the correctness or validity of the system to be verified through testing or formal methods – the system must be implemented before it can be tested. More advanced approaches, e.g., Executable UML (ExUML) or Model Driven Architecture (MDA) [1], allow models to be simulated, i.e., provide simulation testing. Purely formal models, e.g., Petri nets, allow formal or simulation approaches to be used for testing, verification, and analysis.

Model and Simulation-Based System Design (MSBD) refers to a set of techniques and tools for developing software systems that are based on formal models and simulation techniques. It aims to improve the efficiency and reliability of development processes, including software system deployment. One way to increase the efficiency and reliability of development processes is to work with high-level languages and models throughout the development process. In traditional system development methodologies, models are typically created in the analysis and design phases and are input in the implementation phase. The system code is implemented manually by reflecting the created models or by transformations. The fundamental problem with model transformations is often the impossibility of a fully automated process and, therefore, the mismatch between models and their implementation. The transformed code needs to be modified manually, and these changes are not fully reflected in the models. However, if we use a formalism that allows us to include parts of the code, the resulting transformed code does not need further modification. The Object Oriented Petri Nets (OOPN) language is one of these formalisms. This paper focuses on transforming models described by the OOPN formalism into the Java programming language.

There are many approaches in the field of code generation. One direction [2]–[4] generates models in the chosen language from UML models, usually from a class diagram. Other work [5] transforms different levels of diagrams. Still, other approaches attempt to transform conceptual models described in, e.g., SysML into simulation models [6]. There are approaches working with simplified variants of UML models (xUML or fUML) from which it is possible to generate the resulting system more precisely [7][8]. However, freely available tools allow only partial output (often, only a skeleton in the chosen language is generated). The approach closest to ours is based on the Network-within-a-Network (NwN) formalism, with which the Renew [9] tool is associated. NwNs combine Petri nets and the Java language, and models are directly translated into Java. Our approach works with Smalltalk, which can be transformed into Java or C++, or we can directly use these languages for inscription. Our goal is to create a more efficient representation of models for deployment on commonly used platforms and languages (Java, C++).

The paper is structured as follows. In Section II, we introduce the basics of the OOPN formalism. Section III describes the basic structure of the OOPN, which is subject to the transformation whose basic principle is described in Section IV. Chapters V and VI discuss the essential element of the transformation, the component, and its runtime in the Java environment.

II. OBJECT ORIENTED PETRI NETS FORMALISM

An OOPN is a set of classes specified by high-level Petri nets [10]. Formally, an OOPN is a triple (Σ, c_0, oid_0) where Σ is the class set, c_0 is the initial class, and oid_0 is the name of the initial object of c_0. A *class* is determined primarily by the object net and the set of method nets. Object nets describe

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2024. **ISBN: 978-1-68558-194-7**

the possible autonomous actions of objects, while method nets describe the reactions of objects to messages sent to them from outside.

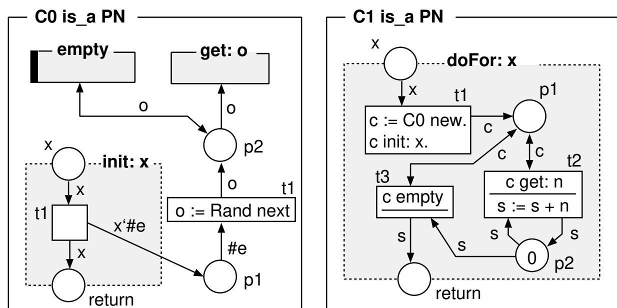

Figure 1. Example of the OOPN model.

An example illustrating the essential elements of the OOPN formalism is shown in Figure 1. Two classes are depicted, *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 the method *doFor:*. An invocation of the method *doFor:* leads to the random generation of *x* numbers and a return of their sum.

Object nets consist of places and transitions. Each place has an initial marking. Each transition has conditions (i.e., inscribed test arcs), preconditions (i.e., inscribed input arcs), guard, action, and postconditions (i.e., inscribed output arcs). *Method nets* are similar to object nets, but each net has a multiplicity of parameter places and the return place. Method nets can access the places of the corresponding object nets to allow running methods to change object states.

Synchronous ports are special (virtual) transitions that cannot be executed independently but are dynamically joined to some other transitions that activate them from their guards via messaging. Each synchronous port contains a set of conditions, preconditions, and postconditions over the places of the corresponding object nets, a guard, and a set of parameters. Thus, synchronous ports combine the concepts of *transitions* (must satisfy preconditions and guards; when a synchronous port is invoked, postconditions are executed) and *method net* (must be invoked from the guard of another transition). The parameters of an activated synchronous port s can be bound to constants or unified with variables defined at the transition level or port that activated the port s.

Negative predicates are special variants of synchronous ports. Their semantics are reversed - the calling transition is executable if the negative predicate is not.

III. Basic Structure

Objects create a network of dependencies through their links, which gradually arise and disappear. On the other hand, the internal nets of an object (object net or method nets) have a clearly defined structure that defines actions and the conditions under which actions can be performed. Each net contains transitions and places. Transitions represent actions whose execution is conditioned on both the existence of the corresponding objects at the entry points and the guarding of the transition. The guard defines the conditions imposed on objects entering the transition. If these conditions are not met, the transition cannot be executed (fired). A transition may be evaluated as feasible and executed for different objects available at the entry points satisfying the guard conditions. Thus, a transition can be viewed as a special kind of component that is dynamically duplicated when the transition is executed and terminates after the last transition action is executed. Thus, transitions, or their execution, represent the internal parallelism of the nets.

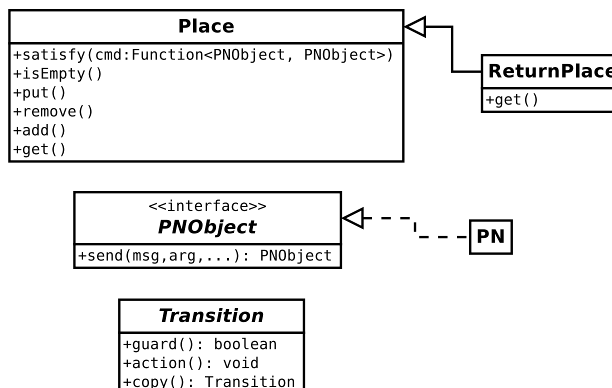

Figure 2. Basic Java classes for OOPN transformation.

Figure 2 shows the basic structure of classes and interfaces required to transform OOPN models into Java. The class Place represents the collection corresponding to a place. In addition to the standard and expected operations for adding, retrieving, and deleting elements, it contains an operation for evaluating a condition placed on the collection's contents. The condition is represented by a function (the Java functional interface Function). When the condition is met, the operation returns an object from the collection that satisfies the condition. The special class ReturnPlace represents the return place of the method nets. It overrides the get method, which is blocking here (it waits for the object to be inserted into the collection, i.e., for the called method to terminate). The meaning of the other elements will be explained in sections IV and V.

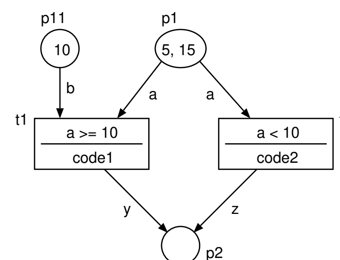

Figure 3. Example: Object net of the class C1.

Consider the simple example in Figure 3. This is an object net of class C1 consisting of two transitions conditioned on places p11 and p1, each transition having in addition its

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

feasibility condition (guard1: $a >= 10$ and guard2: $a < 10$). The result of each transition execution is placed at place p2. Thus, executing this net produces a copy of component t1 (for binding $a = 5$ and $b = 10$) and a copy of component t2 (for binding $a = 15$).

IV. Structure Tranformation

A view of the transition as a component can be used to transform the model into a programming language, in this case, Java.

```
public class C1 extends PN {
    protected Place p11;
    protected Place p1;
    protected Place p2;
    public C1() {
        p11 = new Place(this);
        p1 = new Place(this);
        p2 = new Place(this);

        class T_1 extends Transition { ... }
        T_1 t1 = new T_1();
        class T_2 { ... }
        T_2 t1 = new T_2();

        t1.precond(p11, p1);
        t2.precond(p1);
        p1.add(5;
        p1.add(15);
        p11.add(10);
    }
}
```

Figure 4. Translation of the OOPN model of class C1 into Java.

For each transition, a class derived from the Transition class is generated, containing methods to verify the input conditions (guard) and a method containing the actual actions of the transition (action). A place corresponds to an unordered collection of objects from which objects can be read and removed, and new objects can be added. The principle of model translation is shown in Figure 4. It presents a basic structure of Java code based on the model example shown in Figure 3. The following section will describe each aspect of the code.

The class is always derived from the PN class, which provides the primary means for object handling and communication. The object net is represented by a parameterless constructor (if a constructor is used in the model, the generated constructor in Java is adapted to this). The object net's places can be considered attributes (object variables) of the object, and their declarations are therefore placed in the member fields space. They are then initialized in the constructor, i.e., an instance of the Place class representing a type of collection is created. As will be shown later, it is through the place, or by inserting objects into the place, respectively, that invoke the check for satisfiability of transition (component) conditions; the place must pass information about the object through the constructor.

```
public PNObject m(PNObject p1) {
    ...
    Place ret = new ReturnPlace();
    ...
    // Transition::action -> ret.put(result);
    ...
    return ret.get();
}
```

Figure 5. Example of the method translation into Java.

The OOPN object method has a structure similar to an object net. It differs in the following aspects. It can have input parameters that are modeled as places in OOPN. However, since only one object can be inserted into a place when the method is invoked, a variable can be used directly in the generated method. The method can also return an object as its result. In the OOPN model, such a return object is placed into a place named return by performing some transition. Thus, the method must wait before placing the object in the return place. A special ReturnPlace class with a blocking get() method is provided for this purpose. The method will wait until at least one object is inserted into the place. An example of the skeleton of the generated method is shown in Figure 5.

V. Component Definition

Figure 6 shows an example of a component generated by the transition. The component takes the form of a class derived from the Transition class. The implementation of the guard and action methods depends heavily on the model. The binding of variables from input places is reflected in the guard method. In this example, the input places are checked to see if they are empty and if there is an object that satisfies the condition given by the guard of the transition t1. If these conditions are met, the corresponding objects are stored in the component variables, and the guard method is terminated successfully. Following the success, the component's copy is then executed.

Because the OOPN language is typeless, the common type of all variables is the PNObject class, and communication, i.e., sending messages, must be done specially. PNObject is the interface implemented by the PN class and, thus, by all OOPN classes. However, we must consider that models also work with other objects (e.g., primitive Java data types and other Java classes). Therefore, we need wrappers for objects of these classes that implement the PNObject interface to ensure compatibility. The messaging is done via a special protocol (see the call message in Figure 6), ensuring proper redirection to the target object.

VI. Component Execution

The question is how to verify the feasibility of transitions, i.e., the execution of component actions. Repeatedly testing the satisfaction of conditions is obviously inefficient and completely inappropriate. For these purposes, the Observer design pattern can be used. Each place knows the transitions (components) whose feasibility it affects. At the moment of

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

```
class T_1 extends Transition {
    private PNObject a;
    private PNObject b;
    public boolean guard() {
        // guard1: a >= 10
        if (p11.isEmpty()) return false;
        if (p1.isEmpty()) return false;
        a = p1.satisfy((o) -> o.send(">=", 10));
        if (a == null) return false;
        b = p11.remove();
        p1.remove(a);
        return true;
    }
    public void action() {
        // code1: y = a + b
        PNObject y = a.send("+", b);
        p2.put(y);
    }
    public Transition copy() {
        T_1 t = new T_1();
        t.a = a;
        t.b = b;
        return t;
    }
}
```

Figure 6. Implementation of generated transition t1.

change (it is sufficient to watch for the addition of an object to the place), it notifies all connected components, which verify their state. Access to these collections must be synchronous, as each component is generally expected to run in its thread, and hence, concurrent access may occur. Since verifying the conditions to trigger a transition action (component) must be an atomic operation, a method similar to event-driven programming can be chosen for synchronization. Each object contains a control thread in which the verification of the conditions of all object transitions, i.e., the object net and the method nets, is performed. Since only these nets can access the object places, this will guarantee exclusive access and atomicity of each verification. The control thread is created and started when an instance of the corresponding class is created, and requests for verification of transition feasibility conditions are only processed in its code. The disadvantage of this approach is that the thread exists even after the object is no longer needed and could be removed from memory.

Another approach is to use a monitor that is implicitly available in Java. When invoking condition validation, the object monitor protects the relevant actions within which the places (whether of object net or method nets) are accessed. The monitor object is passed by the constructor when creating an instance of the Place class. A code sample is shown in Figure 7. Each registered transition for which a given place is an input condition is tested for feasibility (called its guard method). If the transition is evaluated as feasible, its action (through the action method) is executed in a separate thread; the executor's service is used via the PNSystem class. Since the component action can be executed simultaneously for different bindings, we need to run the action method of the component copy with

```
void add(PNObject obj) {
  synchronized(monitor) {
    Integer c = content.get(obj);
    c = (c != null) ? c + 1 : 1;
    content.put(obj, c);
    for (Transition t : observers) {
      if (t.guard()) {
        Transition tt = t.copy();
        PNSystem.execute(() -> tt.action());
      }
    }
  }
}
```

Figure 7. The class Place, method add(PNObject).

the current binding in the thread. The copy method is used for this purpose.

VII. Conclusion

This paper aimed to outline the possibilities of transforming the models described by the OOPN formalism into Java. The resulting code does not need to be further modified because the original model allows the use of the code and also objects from the target environment (in our case, Java). The basic principle is quite simple. However, the efficiency of the translated code depends on the analysis of transitions and appropriate optimization techniques. For example, the place corresponding to the input parameter of a method does not need to be generated as a collection because it can contain at most one object. For the same reason, a dependent transition can be executed almost once.

If we include the declaration of [10] types in the OOPN model or automated type derivation, it is possible to replace the generic PNObject type with a specific type in the generated code and thus interact with objects directly by sending messages.

Acknowledgment

This work has been supported by the internal BUT project FIT-S-23-8151.

References

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

[2] T. Hussain and G. Frey, "UML-based Development Process for IEC 61499 with Automatic Test-case Generation," in *IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2010.

[3] C. A. Garcia, E. X. Castellanos, C. Rosero, and Carlos, "Designing Automation Distributed Systems Based on IEC-61499 and UML," in *5th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2017, pp. 61–68.

[4] I. A. Batchkova, Y. A. Belev, and D. L. Tzakova, "IEC 61499 Based Control of Cyber-Physical Systems," *Industry 4.0*, vol. 5, no. 1, pp. 10–13, November 2020.

[5] S. Panjaitan and G. Frey, "Functional Design for IEC 61499 Distributed Control Systems using UML Activity Diagrams," in *Proceedings of the 2005 International Conference on Instrumentation, Communications and Information Technology ICICI 2005*, 2005, pp. 64–70.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-194-7

[6] G. D. Kapos, V. Dalakas, A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos, “Model-based system engineering using SysML: Deriving executable simulation models with QVT,” in *IEEE International Systems Conference Proceedings*, 2014, pp. 531–538.
[7] F. Ciccozzi, “On the automated translational execution of the action language for foundational uml,” *Software and Systems Modeling*, vol. 17, no. 4, p. 1311–1337, 2018, doi: 10.1007/s10270-016-0556-7.
[8] E. Seidewitz and J. Tatibouet, “Tool paper: Combining alf and uml in modeling tools â an example with papyrus,” in *15th Internation Workshop on OCL and Textual Modeling, MODELS 2015*, pp. 105–119, [retrieved: August, 2024]. [Online]. Available: http://ceur-ws.org/Vol-1512/paper09.pdf
[9] L. Cabac, M. Haustermann, and D. Mosteller, “Renew 2.5 - towards a comprehensive integrated development environment for petri net-based applications,” in *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*, 2016, pp. 101–112. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-39086-4_7
[10] R. Kočí and V. Janoušek, “The Object Oriented Petri Net Component Model,” in *The Tenth International Conference on Software Engineering Advances*. Xpert Publishing Services, 2015, pp. 309–315.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org