On the Keeping Models in the System Design and Implementation

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*—An essential criterion for software design and implementation is the effectiveness of requirements specification, development, and verification. One possibility is the use of high-level models and languages. A particular disadvantage is the need to transform models into a production environment, either manually or automated. In both cases, the link to the original models is often lost, degrading their usability in the future. In this paper, using a demonstration example, we will look at the possibility of modeling requirements using Object-Oriented Petri Nets (OOPN) and then transforming them into Java to maintain the model's and implementation's correlation. We will discuss the automation of this process and possible ways to increase efficiency.**

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

I. Introduction

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 throughout the development process. The fundamental problem with model transformations is often the impossibility of a fully automated process and, therefore, the mismatch between models and their implementation. In this paper, using a demonstration example, we will look at the possibility of modeling requirements using Object-Oriented Petri Nets (OOPN) and then transforming them into Java to maintain the model's and implementation's correlation. We will discuss the automation of this process and possible ways to increase efficiency.

There are many approaches to code generation. First, the generation of models in the chosen language from UML models [1]–[3], the transformation of different levels of diagrams [4], or the transformation of conceptual models described, e.g., in SysML into simulation models [5]. Second, more accurate code generation from simplified variants of UML models (xUML or fUML) [6][7]. The biggest pitfall of these approaches at the moment is tool support. Freely available tools often fail to exploit the full potential of the underlying principles. Our closest approaches are probably the Network-within-a-Network (NwN) formalism and the associated tool Renew [8]. Like us, NwN combines Petri nets and the Java language, and the models are directly translated into Java. Nevertheless, our approach works not only with one language, but we can combine Smalltalk, Java, or C++, including directly writing the code within models. We aim to create a system and tool for more efficient modeling and model deployment, including code generation on languages like Java and C++.

The paper is structured as follows. In Section II, we introduce the basics of the OOPN formalism. Section III introduces the demonstration example and describes the basic principles of modeling components and layers based on the Discrete Event System Specification (DEVS) formalism. Section IV describes the way we can move from DEVS like components to objects. Section V discusses possibilities of code generation in two ways – unsupervised and sipervised.

II. Object Oriented Petri Nets Formalism

An OOPN is a set of classes specified by high-level Petri nets [9]. 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 the possible autonomous actions of objects, while method nets describe the reactions of objects to messages sent to them from outside.

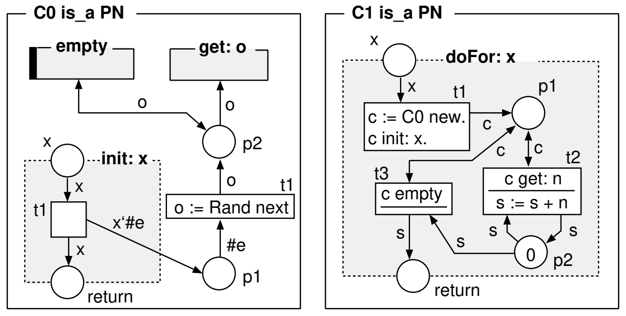

Figure 1. Example of the OOPN model.

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.

III. Requirements Modeling

This section presents a demonstration example and the essence of requirements modeling using the OOPN and DEVS formalisms.

A. Demonstration Example

Let us start with the following example, which is inspired by the simple game LightBulb. First, let's give the basic text description. The game board consists of fields that are either empty, contain connections (only the edges of a square can be connected), an energy power, or a light (bulb). Connectors can connect two, three, or four edges. There is just one source and at least one bulb in the game. The player can rotate each field 90 degrees to the right. At the beginning, the fields are rotated so there is no connection between the power and the bulbs. The game's goal is to rotate the boxes so that the source connects with all the bulbs and thus lights them up. If a field is energized (connected to the power), indicate this by changing color. We will focus here on defining the behavior of each field.

B. Components and Layers in the Model

In the specification and design, we will assume that a field is a component that operates on specific input values and passes information about changes through outputs. It will settle the initial design and reasoning over the requirements and their modeling. We can combine the OOPN (behavioural model) and DEVS (structure and component model) formalisms for these purposes. We start specifying a field as a DEVS component with four input and four output ports. Each input and output pair represents the information transfer between fields adjacent to the corresponding edge, as shown in Figure 2.

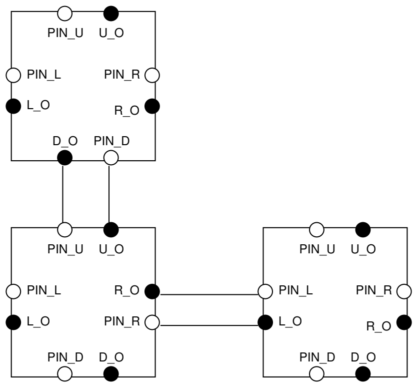

Figure 2. Component model using DEVS formalism.

The behaviour will be defined using the OOPN formalism, where we will create so-called layers. Each layer represents a separate functionality, which can then be modeled as part an object net or a method net. Working with layers allows us to structure behavior better and manipulate the distribution of responsibilities.

C. Initial Requirements Model and Layers

In specifying the requirements, we will start from the elements that follow from the specification, model them using the OOPN formalism, and gradually reveal other essential components.

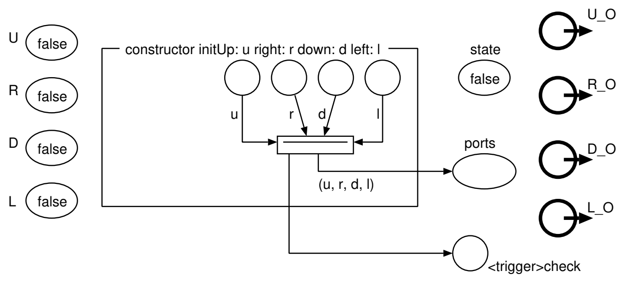

Figure 3. Initial model of the Field component.

The basic definition assumes that each field knows whether an edge is part of a connection. The information is stored in the place ports and can be modeled by a sequence of true/false values indicating whether the edge is connected. The sequence always starts with the top edge and proceeds clockwise. For example, the sequence $(\mathrm{true}, \mathrm{true}, \mathrm{false}, \mathrm{false})$ corresponds to the left bottom field shown in Figure 2. The field stores information about the surrounding fields (modeled as U, R, D, and L places) and informs the surrounding fields of the change (modeled as U_O, R_O, D_O, and L_O output ports). The information indicates whether the field is connected to power. Finally, the place state indicates whether this field is under the power (values true or false). Figure 3 shows the basic model of the Field component. The model is initialized by the constructor $\mathrm{initUp{:}right{:}down{:}left{:}}$.

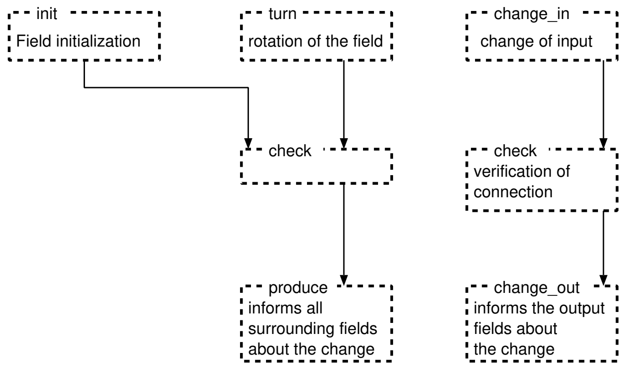
init
Field initialization
turn
rotation of the field
change_in
change of input
check
check
verification of connection
produce
informs all surrounding fields about the change
change_out
informs the output fields about the change

Figure 4. The model flow including layers.

Depending on the task, we can distinguish three basic actions – initialization, rotation, and reaction to a change coming from the surroundings. Figure 4 depicts the basic flows. When the corresponding event is triggered (i.e., the trigger place receives a token), the field is always checked to see whether the field is connected to the energy source, i.e., whether the triggered action caused the change. However, the subsequent actions differ according to the originator, so that we can find two basic flows. In the case of initialization and rotation, we have to inform all the surrounding fields, since a link between fields may have been created or broken. In the case of a change coming from the surroundings, we only inform the fields connected through existing links of the change, if any.

D. Basic Layers

Now, we will look at the model's layers. First, the check layer, which is shown in Figure 5. The model clearly shows how the new state is evaluated.

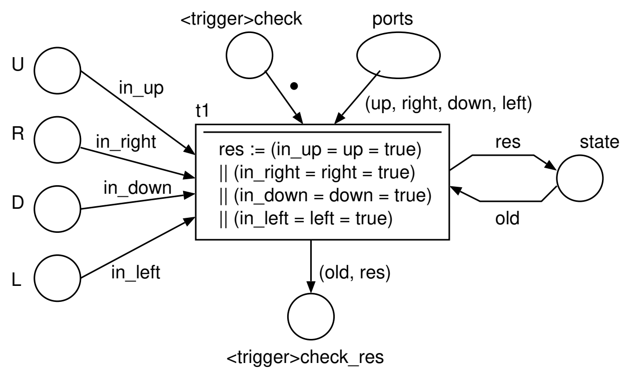

Figure 5. The check layer.

The evaluation is based on the knowledge of the state of the surrounding fields (places U, R, D, and L) and whether the corresponding edge contains a connection to the corresponding fields (see the place ports). If at least one edge containing a connection is true, this field is connected to the energy power. The new value has been rewritten in the state place. The execution of the layer is conditioned by the token in the place *check*, and termination is indicated by inserting the previous and new state pairs in the place check_res.

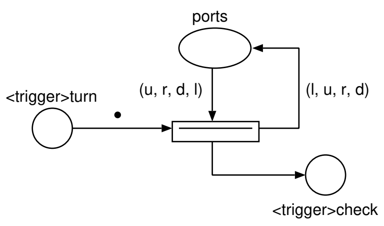

Figure 6. The turn layer.

The next layer is turn, which is captured in Figure 6. The principle of modeling this functionality is evident from the model – the original sequence is removed from the place ports and new sequence, which rotates one position to the right, is inserted back. The execution of the layer is conditioned by the token in the trigger place turn. The termination can be indicated by the addition of an exit place, similar to the check layer. The necessity of such an addition will become apparent during model creation.

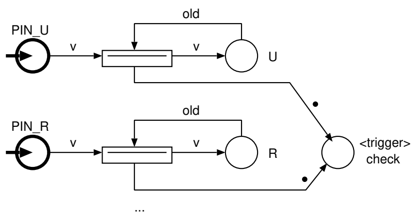

Figure 7. The change_in layer.

The next layer is change_in, which is captured in Figure 7. This layer responds to a change in an input port value and ensures that the contents of the corresponding places are changed. Ports are modeled by special places; see, e.g., PIN_U in Figure 7. If the new value is accepted from the surrounding, it is placed to the input port. The layer updates field's information about the surrounding and activates the check layer by inserting a token in the place check. The model assumes that the change occurs at exactly one input port at a time.

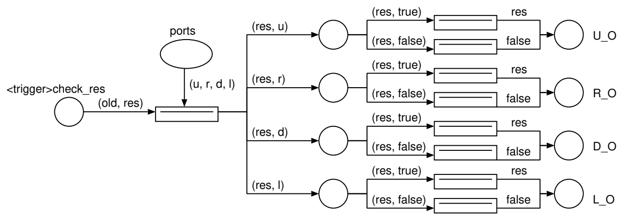

Figure 8. The produce layer.

The next layer is produce, which is captured in Figure 8. This layer is activated by inserting a pair of values of the original and new value of the field state. The layer ensures that the new value is distributed to the surrounding patches through the output ports. However, the insertion of the new value is conditional on a connection on the corresponding edge. If there is no connection, it informs the connected patch of the false state (if there was a connection in the previous state, the connected patch must process this change).

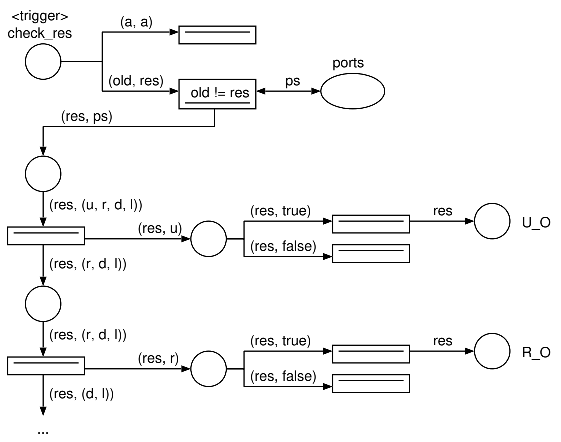

Figure 9. The change_out layer.

The next layer is change_out, which is captured in Figure 9. This layer is fundamentally similar to the layer produce, it captures an alternative modeling option. First, it checks whether the state has changed. If so, all surrounding fields are informed, i.e., the new value is inserted into the appropriate output port if it exists on that link edge. If there is no connection on the edge, no value is inserted (the field is not informed).

IV. Transformation of DEVS Components to Objects

So far, we have considered the model of the field as a DEVS component that communicates with its surroundings through ports. However, we need a conventional approach for use in classical programming languages and environments, i.e., communication via messaging. At the same time, in the modeling, we worked with a sequence of logical values at the port location, which determined which edges contained the connector. It carries specific modeling implications, e.g., repetitive capture of the same functionality over different edges, since the result must always be placed in a different component output port. This section illustrates two steps. First, the edges are named for more flexible handling, and then the DEVS component is transformed into an object component.

A. Ports identification

For our simple example, we choose naming using symbols that bind to the pair $(\text{name}, \text{exist})$ in the place ports.

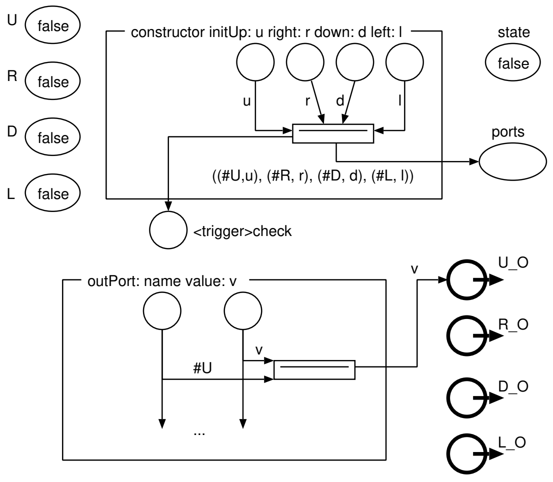

Figure 10. Modification of the init layer.

Figure 10 shows a preview of the change at the init layer. At the same time, we introduced the outPort:value: method, which inserts the specified value into the output port identified by name. We make similar modifications for the check and turn layers (see Figures 11 and 12).

B. Replacement of Ports

We will show more substantial modifications to the produce layer. Since we have the output ports named, we can use the concept of foreach as shown in Figure 13. The basic idea

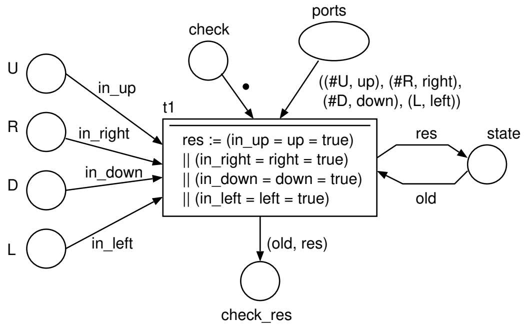
check
ports
U
in_up
((#U, up), (#R, right), (#D, down), (L, left))
t1
R
in_right
res := (in_up = up = true)
|| (in_right = right = true)
|| (in_down = down = true)
|| (in_left = left = true)
res
state
D
in_down
old
L
in_left
(old, res)
check_res

Figure 11. Modification of the check layer.

of the foreach loop is based on list processing in the Prolog language.

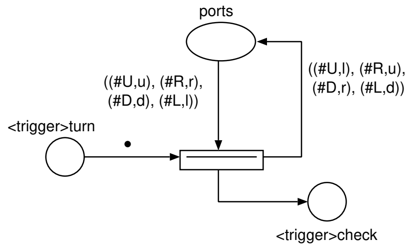
ports
((#U,u), (#R,r), (#D,d), (#L,l))
((#U,l), (#R,u), (#D,r), (#L,d))
<trigger>turn
<trigger>check

Figure 12. Modification of the turn layer.

Let us return to the produce layer (Figure 13). We build on the original solution, but instead of inserting a value into a specific output port, we call the outPort:value: method. This evaluation is done only once for all edges stored in the place ports. A similar modification could be made for the check_out layer.

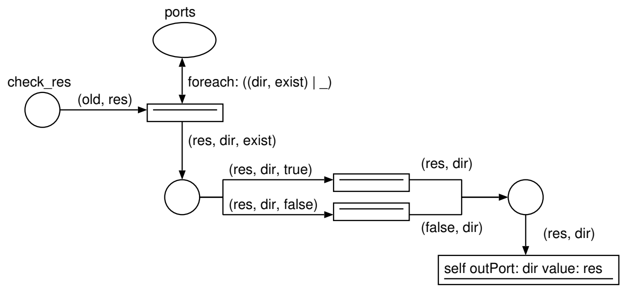

Figure 13. Modification of the produce layer.

Finally, we replace ports with methods or method calls. We generate a corresponding method for each input port with the same name and one argument. Instead of passing data through DEVS components, objects will send messages to each other. An example of changing the PIN_U input port to a method is shown in Figure 14.

Output ports are replaced by calling the corresponding method. For instance, for the output port U_O, it is necessary to call the method PIN_D:, because the output of the up field corresponds to the down input of the connected field (see the

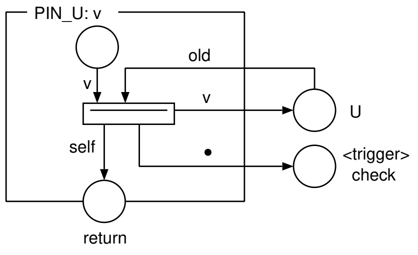

Figure 14. Replacing the input port with the method.

DEVS component model in Figure 2). An example is shown in Figure 15.

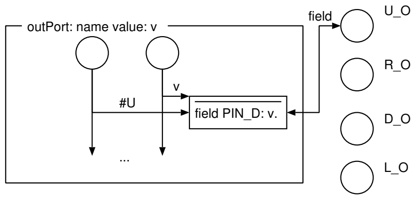

Figure 15. Replacing the output port with the method call.

Figure 16 schematically depicts the resulting OOPN model – places, methods, and basic layers initiated by trigger places.

V. Code Generation

This section presents the possible outputs of the model transformation into Java. We build on the work of [10]. Due to the generality of the OOPN formalism, the fundamental transformation mechanism is cumbersome (*unsupervised generation*), but introducing some additional information can make code generation more efficient (*supervised generation*). This information can be supplied manually, or it can be derived by automated analysis of the model. We will present examples of generated code for only one part of the model. In both cases, we obtain executable code that differs in complexity and efficiency.

A. Basic Framework Classes

The created OOPN models, which correspond to the principles mentioned so far, can be automatically translated into Java. The resulting class system needs a basic framework prepared for these purposes [10]. Figure 17 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. The OOPN class is always derived from the PN class, which provides the primary means for object handling and communication. The object net is represented by the constructor. 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. 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.

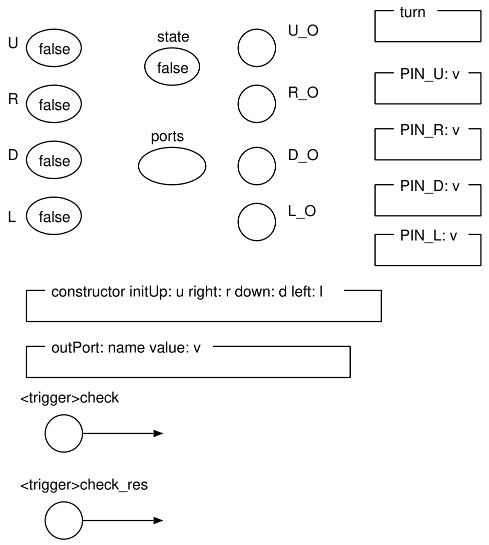

Figure 16. Model of the class Field overview.

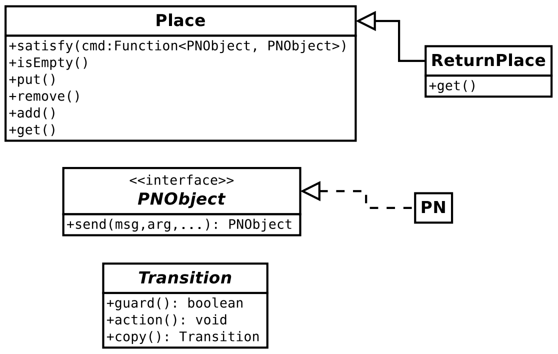

Figure 17. Basic Java classes for OOPN transformation.

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. 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.

B. Unsupervised Generation

As mentioned, we will demonstrate the transformation (code generation) capabilities only on selected parts of the model. The model consists of a single class $Field$. Figure 18 shows the generated code for the model layer captured in Figure 16 in the basic (unsupervised) version.

All variables and values are typed as class $PNObject$. A special class $PNList$ is used to implement the list of values. This figure does not capture the whole listing; it is only an outline of the generated code.

```
public class Field extends PN {
    protected boolean state;
    protected List<List<Dir, Boolean>> ports;
    protected boolean U, R, D, L;
    protected boolean U_O, R_O, D_O, L_O;
    public enum Dir {U, R, D, L};
    public C1(boolean u, boolean r, boolean d,
              boolean l) {
        state = false;
        ports = new ArrayList<>();
        ports.put(new ArrayList<>(Dir.U, u));
        ports.put(new ArrayList<>(Dir.R, r));
        ports.put(new ArrayList<>(Dir.D, d));
        ports.put(new ArrayList<>(Dir.L, l));
        ...
    }
}
```

Figure 19. Supervised translation of the class Field into Java.

C. Supervised Generation

For supervised generation, we use the constraints introduced in [9], which allow us to define different constraints on models. The constraints can be defined manually or derived by analyzing the model or its simulated run [11]. This analysis finds the following constraints on the model under consideration.

```
public class Field extends PN {
    protected Place state;
    protected Place ports;
    protected Place U, R, D, L;
    protected Place U_O, R_O, D_O, L_O;
    public C1(PNObject u, PNObject r, PNObject d,
              PNObject l) {
        state = new Place(this);
        ports = new Place(this);
        fields = new Place(this);
        inputs = new Place(this);

        state.add(false);
        PNlist lst = new PNList();
        lst.add(new PNList("#U", u));
        lst.add(new PNList("#R", r));
        lst.add(new PNList("#D", d));
        lst.add(new PNList("#L", l));
        ports.add(lst);
        ...
    }
}
```

Figure 18. Unsupervised translation of the class Field into Java.

$\text{context Field::state: Boolean}$
$\text{context Field::ports: OrderedList}$
$\text{context Field::ports element: OrderedList (Dir, Boolean)}$
$\text{context Dir: enum}(\mathrm{U}, \mathrm{R}, \mathrm{D}, \mathrm{L})$
...

The generated code can be simplified and streamlined based on the defined constraints, as shown in Figure 19.

VI. Conclusion

This paper presented the possibilities of modeling requirements using OOPN formalisms (for behavior definition) and DEVS-like components (for structure description). The model can then be gradually transformed into a more efficient form and a programming language (currently Java). The essential feature we want to achieve is that the resulting code does not need to be further modified, because the original model allows the use of code and objects from the target environment. Thus, all changes and modifications occur at the model level.

In the future, we want to focus on fully automated constraint derivation over the model (while retaining the possibility of manual intervention) and automated support for model modifications. For these purposes, we plan to explore the possibilities of involving artificial intelligence, particularly large language models (LLMs). It also assumes tool support, which we will continue to work on.

Acknowledgment

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

References

[1] T. Hussain and G. Frey, “UML-based Development Process for IEC 61499 with Automatic Test-case Generation,” in *2006 IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2006, pp. 1277–1284.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] E. Seidewitz and J. Tatibouet, “Tool paper: Combining alf and uml in modeling tools: An example with papyrus,” in *15th Internation Workshop on OCL and Textual Modeling, MODELS 2015*, pp. 105–119, [retrieved: June, 2025]. [Online]. Available: http://ceur-ws.org/Vol-1512/paper09.pdf

[8] 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

[9] 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.

[10] R. Kočí, “On the object oriented petri nets model transformation into java programming language,” in *The Nineteenth International Conference on Software Engineering Advances, ICSEA 2024*. Xpert Publishing Services, 2024, pp. 38–42.

[11] R. Kočí and V. Janoušek, “Tracing and Reversing the Run of Software Systems Implemented by Petri Nets,” in *ThinkMind ICSEA 2018, The Thirteenth International Conference on Software Engineering Advances*. Xpert Publishing Services, 2018, pp. 122–127.