Incorporating Petri Nets into DEVS Formalism for Precise System Modeling

Radek Kočí and Vladimír Janoušek

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

***Abstract*—Modeling and simulation are part of software development because of their ability of system abstraction and validation. One of the research motivation is oriented towards more interactivity during system requirements modeling and a possibility to investigate models under real condition. To achieve this goal, the system has to be modeled precisely. There is a lot of suitable paradigms—the paper concentrates especially on Object Oriented Petri Nets (OOPN) and Discrete-Event Specification (DEVS) formalisms. OOPN constitute an abstract formalism allowing a natural description of parallelism, synchronization and non-determinism. The formalism of DEVS constitutes a basis for a theory of modeling and simulation and can be considered as a basic platform for multi-paradigm system design. Therefore, the formalism of OOPN has been incorporated to the formalism of DEVS.**

Keywords–Object Oriented Petri Nets; DEVS; modeling; simulation; interconnection.

I. Introduction

Modeling plays an irreplaceable role in the software community, but its meaning is perceived in different ways. Its primary task is to better present the domain problem solution, to facilitate understanding of domain elements and, thus, simplify the process of system analysis. In this understanding, the model usually takes the form of static figures, which are no longer used after analysis and design, eventually they serve only as a starting point for implementation. However, the implementation gradually moves away from these models and the resulting product no longer corresponds to these models. At this stage, tool-based testing or analysis cannot be provided because the used models do not have a sufficient level of precision in the specification of requirements. Some specified elements need to be verified at a later stage of development, or it is necessary to implement a prototype on which these actions can already be performed. Both approaches extend the time required to properly validate requests.

To model and analyze systems under real conditions during the process of creating requirements and design, it is desirable that these formal models can be linked to parts of the implementation that would be subject to the same way of running control. In order to accurately determine point the system diverts from expected behavior, we should be able to stop the simulation and analyze it. We need to have such formalisms that allow to control model execution.

The most widespread modeling tool is Unified Modeling Language (UML) and its variants or adds that refine models (e.g., Object Constraint Language—OCL [1]) to allow their simulation (e.g., Executable UML). Generally, these methods can be called executable modeling [2]. However, the used formalisms usually lack an inherent relationship between graphical representation of models and the precise specification of system behavior. There are approaches that attempt to describe selected UML models in a fully formal way, such as Foundational Subset for Executable UML Models (fUML) [3], [4], which can specify a part of UML models by formal description. On the other hand, this blurs the advantage of UML, namely graphic notation.

The concept of model based design and executable modeling is mainly applied to cyber physical systems [5] and is not very widespread for system design and implementation. This paper is a follow-up to the work on application of formal models for specification and design of software systems [6], [7], which concentrates on DEVS and Petri nets formalisms. DEVS is a systems specification formalism developed by B. Zeigler [8]. It constitutes a basis for theory of modeling and simulation and can be considered as a basic platform for multi-paradigm modeling and simulation [9]. Object Oriented Petri Nets (OOPN) constitute a formalism suitable for structure modeling with graphical notation [10].

If we want to integrate the created models into the real environment, we must have a suitably adapted interface to the system in which the considered real environment is implemented. Implementation means its simulation, control of simulated or real components, communication with a database or other systems, etc. The formalism of OOPN allow to incorporate executive code into a model—the code is placed in transition actions. Such a solution makes it very difficult to adopt already existing code, complicates the simulation and analysis of the model and may not be completely conformist from the designer's point of view. Another solution is to select the appropriate interface and delegate code execution to formalism, which allows direct code incorporation while adopting a run-time management method. DEVS is such a formalism, but it becomes less transparent for larger systems (there is no other form of graphical presentation other than blocks). Therefore, this paper deals with an incorporation of OOPN and DEVS formalisms.

The paper is organized as follows. First, we briefly present formalisms of OOPN and DEVS (Section II). Section III takes simulation control of both formalisms into account. Section IV deals with the problem of incorporating OOPN into DEVS-based framework. The OOPN and DEVS incorporation is demonstrated on simple example in Section V. At the end, we discuss results and benefits of presented solution.

II. Used Formalisms

This section briefly introduces formalisms of OOPN and DEVS that are taken into account in the paper.

A. Formalism of Object Oriented Petri Nets

Formally, an OOPN is a tuple $\Pi = (\Sigma, \Gamma, VAR, CONST, c_0, oid_0)$, where Σ is a system of classes, Γ is a system of objects, VAR is a set of variables, $CONST$ is a set of constants, c_0 is an initial class, and oid_0 is the name of an initial object instantiated from c_0.

Σ contains sets of structural elements, which constitute classes. It comprises classes $CLASS$, net elements (places P and transitions T), class elements (object nets $ONET$, method nets $MNET$, synchronous ports $SYNC$, negative predicates $NPRED$), and message selectors MSG. We denote $NET = ONET \cup MNET$.

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. Synchronous ports are special transitions, which cannot fire alone but only dynamically fused to some other transitions.

Elements from $CLASS$ and NET describe a structure of simulation model and have to be instantiated to simulate the model. For example, the instance of initial class c_0 has to be created with the object identifier oid_0. At the same time, there are created instances of object net. If the message is sent to the object, an instance of the method net is created.

Let us define Γ as a structure containing sets of object identifiers OID, and method net instance identifiers MID. We denote $ID = OID \cup MID$. Object net is strictly connected with the class, so that we can identify its instance by object identifier. We also define universe U as the set of tuples of constants, classes, and object identifiers. The set of all bindings of variables used in OOPN is then defined as $BIND = \{b \mid b : \mathit{VAR} \longrightarrow U\}$.

A state of a running OOPN model has the form of *marking* of net instances. Marking is represented as the multiset of token elements. An element of transition marking has a form of cartesian product $ID \times T \times BIND$, where $nid \in ID$ represents the identifier of the method or object net instance, $t \in T$ is a static representation of transition in the net instance nid, and $b \in BIND$ is one element of bound variables. An element of place marking has a form of cartesian product $ID \times P \times U$, where $nid \in ID$ represents the identifier of the net instance, $p \in P$ is a static representation of place in the net instance nid, and $u \in U$ is one element of place content. A state s is then define as an item of multiset $s \in [(ID \times T \times BIND) \cup (ID \times P \times U)]^{MS}$.

Evaluation of transition fireability is based on high-level Petri net evaluation—a transition is *fireable* for some binding of variables, which are present in the arc expressions of its input arcs and in its guard expression, if there are enough tokens in the input places with respect to the values of input arc expressions and if the guard expression for the given binding evaluates to true.

B. Formalism of DEVS

DEVS is a formalism which can represent any system whose input/output behavior can be described as sequence of events. DEVS is specified as a structure

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where

X is the set of input event values,
S is the set of state values,
Y is the set of output event values,
$\delta_{int} : S \longrightarrow S$ is the internal transition function,
$\delta_{ext} : Q \times X \longrightarrow S$ is the external transition function,
$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the set of total states,
e is the time passed since the last transition,
$\lambda : S \longrightarrow Y$ is the output function,
$ta : S \longrightarrow R^{+}{}_{0,\infty}$ is the time advance function.

At any time, the system is in some state $s \in S$. If no external event occurs, the system is staying in state s for $ta(s)$ time. If elapsed time e reaches $ta(s)$, then the value of $\lambda(s)$ is propagated to the output and the system state changes to $\delta_{int}(s)$. If an external event $x \in X$ occurs on the input in time $e \leq ta(s)$, then the system changes its state to $\delta_{ext}(s, e, x)$.

This way we can describe atomic models. Atomic models can be coupled together to form a coupled model CM. The later model can itself be employed as a component of larger model. This way the DEVS formalism brings a hierarchical component architecture. Sets S, X, Y are obviously specified as structured sets. It allows to use multiple variables for specification of state and we can use input and output ports for input and output events specification, as well as for coupling specification. A lot of extensions and modifications of the original DEVS has been introduced, such as parallel DEVS [11] or dynamic structure DEVS [12] and a lot of simulation frameworks has been developed.

III. Simulation Control

This section describes basic concepts of simulation control for OOPN and DEVS formalisms.

A. DEVS Simulation

DEVS simulation is a structure $SIM_D = (DM, \tau_D, D)$, where DM is a DEVS model, $\tau_D \in \mathbb{N}$ is a model time of DEVS simulation, and D is a set of *solvers*. DEVS model consists of coupled or atomic subcomponents, each such a component is controlled by its *solver*. The simulation of DEVS model is controlled by special *root solver*. The simulation control can be described as shown in Figures 1, 2, and 3.

```
$\tau_D \leftarrow 0$
$ta \leftarrow 0$
while $ta \neq \infty$ do
    call solver on $root\ model$
    $ta \leftarrow ta$ from $root\ model$
    if $ta \neq \infty$ then
        $\tau_D \leftarrow \tau_D + ta$
    end
end
```

Figure 1. DEVS Root Solver Control

The root solver (Algorithm 1) works in cycle until the time advance ta gets infinity. In each step, it calls a solver of root model and updates the model time τ_D. If there is no component which is able to do a step, the ta is set to infinity.

```
1 $DM_i$ is a set of subcomponents of the component $i$
2 $ta = \infty$
3 foreach $m \in DM_i$ do
4     call solver on $m$
5     propagate event values from output ports of $m$ to
        connected input ports of subcomponents from
        $DM_i$
6     $t_m \leftarrow ta$ from component $m$
7     $ta \leftarrow min(ta, t_m)$
8 end
```
Figure 2. DEVS Coupled Solver Control

The solver of coupled component (Algorithm 2) goes through all subcomponents. For each subcomponent, the solver calls a subcomponent solver and propagates event values from its output ports to input ports of connected subcomponents. At the end, the solver sets ta to be a minimum value of ta of all subcomponents.

```
1 $m$ is an atomic DEVS model
2 if $ta = 0$ then
3     execute the output function $\lambda$ on $m$
4     execute the internal function $\delta_{int}$ on $m$
5 else
6     if an external event $x \in X$ occurs then
7         execute the external function $\delta_{ext}$ on $m$
8     end
9 end
10 $ta \leftarrow ta$ from component $m$
```
Figure 3. DEVS Atomic Solver Control

The solver of atomic component (Algorithm 3) tests two conditions. If its $ta = 0$, the output and internal functions are executed. If $ta > 0$ (the component waits for the event) and any external event occurs (an input port contains an event value), the external function is executed. Finally, a new value of ta is obtained.

B. OOPN Simulation

To simulate OOPN, we first extend previously established definitions. The system of objects $\Gamma = (OID, MID, PID, FTID)$ is extended to identifiers of place instances PID and fired transition instances $FTID$. We denote $ID = OID \cup MID \cup PID \cup FTID$. Let define a relation $@ \subseteq ID \times CLASS \cup NET \cup P \cup T$, which represent relationships *is an instance of*. For example, $(a, C1) \in @$ means that a is an instance of the element named `C1`. We will write this relation in the form `a@C1`. If the instance identifier is not important, we will type only `@C1`.

OOPN simulation is a structure $SIM_\Gamma = (\Pi, \tau_\Gamma, CAL)$, where Π is the system of OOPN classes and objects, $\tau_\Gamma \in \mathbb{N}$ is a model time, and $CAL = \{(t, e) \mid t \in \mathbb{N} \wedge e \in FTID\}^{MS}$ is a calendar represented by multiset of timed events.

The transition $t \in T$ can be fireable for any of possible bindings $\mathcal{P}(BIND)$. If the transition t is fired for the binding $b \in \mathcal{P}(BIND)$, three possibilities can occur—(a) the fired transition is completed immediately (it contains simple action), (b) the fired transition will wait for specified time, or (c) the fired transition will wait for called method finishing. The possibilities (b) and (c) imply that the *fired transition* $ft \in FTID$ is created.

Three kinds of events can occur during the simulation—fireable transition (it can be fired), fired transition (it can be complete, i.e., the method called from this transition finished), and timed fired transition (the transition waits for specified time). The first and second events are called *executable events* and the third one is called *timed event*.

Let define $objevents_\Gamma : OID \rightarrow \mathcal{P}(T \cup FTID)$ determining *executable events* over all nets of the object, ϑ determining a least time of event from the calendar, where

$$\vartheta(C) = \begin{cases} \infty, & C = \emptyset \\ t, & \exists (t, e) \in C \wedge \forall (t_i, e_i) \in C : t \leq t_i \end{cases}$$

and $events_\vartheta(C) = \{e \mid (t, e) \in C \wedge t = \vartheta(C)\}$ determining a set of *timed events* having least time in the calendar.

The simulation control is described in Algorithm 4. While there is possible to do a step ($activity = true$), the simulator calls one simulation step.

```
1 $activity \leftarrow true$
2 while $activity = true$ do
3     call $step$
4 end
```
Figure 4. OOPN Simulation Control

The simulation step is described in Algorithm 5. First, it obtains a set of objects having at least one *executable event*. If this set in not empty, the simulator selects an event from each such an object and fires it. Firing events means that the transition is fired and completed, or the transition is only fired (so that the fired transition $ft \in FTID$ arises), or the fired transition is completed. Second, if the set of objects is empty, the simulator obtains a set of *timed events*. If this set is not empty, it sets a model time τ_Γ to the value of ϑ and releases all events waiting at the time ϑ. Releasing timed events means that they are removed from calendar and becomes *executable events*. Third, if there is no event in the calendar, the simulation is finished ($activity \leftarrow false$).

IV. EMBEDDING OOPN INTO DEVS

There are several techniques to integrate various formalisms. The most common ones are *combination*, *mapping*, and *wrapping*. The combination derives *a new formalism* from already existing ones by their combination, e.g., DEV&DESS [13] or Hybrid Petri nets [14]. The *mapping* approach maps formalisms to supporting one so that just the supporting formalism is interpreted. The *wrapping* approach connects simulators of different formalisms so that each model is interpreted by its simulator and simulators communicate with each other by means of a compatible interface. It is advantageous if one of the formalisms can be use as a control simulator for all other formalisms. Since DEVS rigorously defines the component interface, it is very suitable to serve as a basic component

```
1  $objs = \{o \in OID \mid objevents_\Gamma(o) \neq \emptyset\}$
2  if $objs \neq \emptyset$ then
3      foreach $o \in objs$ do
4          $e \leftarrow$ select an item from $objevents_\Gamma(o)$
5          fire $e$
6      end
7  else
8      $ev = events_\vartheta(CAL)$
9      if $ev \neq \emptyset$ then
10         $\tau_\Gamma \leftarrow \vartheta(CAL)$
11         foreach $e \in ev$ do
12             release event $e$
13         end
14     else
15         $activity \leftarrow false$
16     end
17 end
```

Figure 5. The OOPN Simulation Step.

platform for multiparadigmatic simulations. Consequently, we have chosen *wrapping* to embed the OOPN formalism to the DEVS formalism.

A. Ports and Places Mapping

Set of input event values X and output event values Y used in the DEVS formalism can be specified as structured sets. It allows to use multiple variables for specification of state and we can use named input and output ports for input and output events specification, as well as for coupling specification. Let us have the structured set $X = (V_X, X_1 \times X_2 \times \cdots \times X_n)$, where V_X is an ordered set of n variables and $X_1 \times X_2 \times \cdots \times X_n$ denotes a value for each member from the set V_X. We can write the structured set as $X = \{(v_1, v_2, \ldots, v_n) | v_1 \in X_1, \ldots, v_n \in X_n)\}$. Members $v_1, v_2, \ldots, v_n$ are called *input ports* for the set X and *output ports* for the set Y.

DEVS components communicate each other through their ports—when a new object is placed to the output port of a component, it is carried to the appropriate input port of the connected component. The way how to relate DEVS with OOPN is to map ports and places.

Let $M_{PN} = (M, \Pi, map_{inp}, map_{out})$ be a DEVS component M which wraps an OOPN model Π, $c_0 \in CLASS$ is an initial class of the model Π, $oid_0 \in OID$ is an initial object of the class c_0, V_X is an ordered set of input ports of the model M, and V_Y is an ordered set of output ports of the model M. Let define $onet_\Sigma : CLASS \rightarrow ONET$ determining an object net of the class, $places_\Sigma : NET \rightarrow \mathcal{P}(P)$ determining a set of places of the net, and $place_\Gamma : ID \times P \rightarrow PID$ determining an instance of a place in the net. We divide places of object net of an initial class c_0 into two groups $P^{inp}_{c0}, P^{out}_{c0} \subseteq places_\Sigma(onet_\Sigma(c_0))$, where $P^{inp}_{c0} \cap P^{out}_{c0} = \emptyset$. Then we can define a mapping of OOPN places into DEVS ports as bijections $map_{inp} : P^{inp}_{c0} \rightarrow V_X$ and $map_{out} : P^{out}_{c0} \rightarrow V_Y$.

Informally, if an OOPN model is defined as a DEVS component, then an object net of initial class defines input and output places. The initial class is instantiated immediately the component is created, and the defined places serve as input or output ports of the component.

B. OOPN Simulation Control Adaptation

In addition to place mapping, the simulation control has to be adapted too. The OOPN simulator has to define functions ta, δ_{ext}, λ, and δ_{int}.

After each step, the simulator checks the time of the next step by the function *timeAdvance* ($t(a)$). It tests three conditions, as shown in Figure 6: (1) if there is at least one executable object, the advance time is 0; (2) if there is at least one timed event with activating time t, the time advance is a difference of t and current model time τ_Γ; (3) if there is at least one value in any place which is mapped as output port, the advance time is 0 (the output function has to be executed to propagate values from output ports); (3) otherwise, time advance is $infinity$.

```
if $\exists o \in OID : objevents_\Gamma(o) \neq \emptyset$ then
    return 0
else
    $t = \vartheta(CAL)$
    if $t \neq \infty$ then
        return $t - \tau_\Gamma$
    else
        if $\exists y \in V_Y : map^{-1}_{out}(y)$ is not empty then
            return 0
        else
            return $\infty$
        end
    end
end
```

Figure 6. timeAdvance ta

The external transition function δ_{ext} is described in Fugure 7. If the component, which wraps OOPN model, receives an external event (new data in its input ports), the function *extTransition* (δ_{ext}) is called. It takes values out from input ports and puts them into mapped places.

```
foreach $x \in V_X$ do
    $p = map^{-1}_{inp}(x)$
    $v \leftarrow$ a value from $x$
    if $v$ is not empty then
        put $v$ into a place $place_\Gamma(oid_0, p)$
    end
end
```

Figure 7. extTransition δ_{ext}

The output function λ is described in Figure 8. If the component, which wraps OOPN model, has any value to be put output ports, it takes values from mapped places and puts them into into appropriate output ports.

The internal function δ_{int} is described in Figure 9. It is modified simulation step from Figure 5. First, the model time τ_Γ is not updated directly by OOPN simulator (see the line 10 in Figure 5), but is set to the model time τ_D of DEVS root solver (see the line 1 in Figure 9). Second, the liveness of simulation is not tested by means of the attribute $activity$, but the function ta. Third, the simulation cycle is subordinate to DEVS root solver, so that the simulation control described in Figure 4 is not in use.

```
foreach $y \in V_Y$ do
    $p = map_{out}^{-1}(y)$
    $v \leftarrow$ a value from $place_\Gamma(oid_0, p)$
    if $v$ is not empty then
        put $v$ into an output port $y$
    end
end
```

Figure 8. outputFunction λ

```
$\tau_\Gamma \leftarrow \tau_D$
$objs = \{o \in OID \mid objevents_\Gamma(o) \neq \emptyset\}$
if $objs \neq \emptyset$ then
    foreach $o \in objs$ do
        $e \leftarrow$ select an item from $objevents_\Gamma(o)$
        fire $e$
    end
else
    $ev = events_\vartheta(CAL)$
    if $ev \neq \emptyset$ then
        foreach $e \in ev$ do
            release event $e$
        end
    end
end
```

Figure 9. intTransition δ_{int}

C. Controlling External Events

In the course of simulation, the events arise and are served inside the model. Nevertheless, DEVS components or OOPN objects can serve as interfaces to the outer world and the incidental events from the world can arise. DEVS controlling uses *se-message*—a mechanism for a notification of the root solver about a state event, which serves as a request for internal transition function. This mechanism is used especially in real-time simulations.

In addition to *internal events* (fireable transitions, fired transitions, and timed fired actions), the simulation of OOPN distinguishes *control event* (serving method nets instantiation and destroying) and *external event*. Because OOPN objects can communicate with objects from the outer world, OOPN is able to work with extended set of classes $ECLASS = CLASS \cup PCLASS$, where $PCLASS$ is a set of classes of product environment. Then, external event represents a message called from object $o@C$, where $C \notin CLASS$. If the message is received, the control event instantiates method net, consequently, new internal events arise, and a signal for starting simulation cycle in the case of $activity = false$ is generated. When the OOPN model is wrapped to DEVS component, the signal for simulation cycle is simply substituted for *se-message*.

V. Demonstration of OOPN embedding

We demonstrate embedding the OOPN formalism to the DEVS formalism on a simple example.

A. Model

The model consists of two atomic components $D1$ and $D2$. Each component has one input port inp and one output port $outp$ ($V_X = \{inp\}$ and $V_Y = \{outp\}$). Component ports are connected as shown in Figure 10 (on the left).

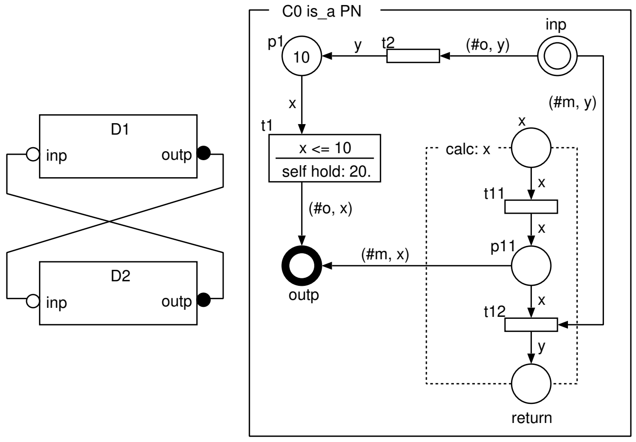

Figure 10. An OOPN-DEVS example.

The DEVS component $D2$ is atomic component, which gets a value x at its input port inp, and puts a value $x+1$ to its output port $outp$. Formally, the component $D2$ defines following functions:

$$\begin{aligned} \delta_{ext}: &\quad x \leftarrow V_X(inp) \\ \lambda: &\quad x \neq \infty \implies V_Y(outp) \leftarrow x+1 \\ \delta_{int}: &\quad x \leftarrow \infty \\ ta: &\quad \begin{cases} 0, & x \neq \infty \\ \infty, & x = \infty \end{cases} \end{aligned}$$

The DEVS component $D1$ wraps an OOPN model Π consisting of the class $C0$ (see the Figure 10 on the right). The OOPN class has an object net and method net $calc$:. The object net consists of places $p1$, inp, and $outp$, and transitions $t1$ and $t2$. The class $C0$ is an initial class c_0 of the model Π and oid_0 is an initial object. The input port inp is mapped to the place inp and the output port $outp$ is mapped to the place $outp$.

B. Simulation without External Events

First, we will investigate what happens if the model simulation starts and nobody will call the method $calc$: as an external event. Possible states of the object oid_0 are shown in Figure 11 in the form of place and fired transition marking—$p1(10)$ means a place $p1$ with one token a number 10, $@t1(x=10)$ means a fired transition $t1$ with bound variable $x = 10$.

Simulation steps of components $D1$ and resp. $D2$ are shown in Figure 12 and 13, respectively. Rows represents component's simulation step, columns contain information about the step: i is step sequence number (over whole simulation), τ_D is model time, s is current state, $\frac{\lambda}{\delta_{ext}}$ represents results of stated functions, δ_{int} is a new state after executing the function δ_{int}, $event$ represents an event which has been fired (transition or fired transition with the binding), and ta is time advance after this step. Since the event, which has been fired, is relevant only for OOPN component $D1$, the column $event$ is not present for the component $D2$.

s	$object\ net\ oid_0$
s_0	$p1(10), inp(), outp()$
s_1	$p1(), inp(), outp(), @t1(x = 10)$
s_2	$p1(), inp(), outp(10)$
s_3	$p1(), inp(), outp()$
s_4	$p1(), inp(11), outp()$
s_5	$p1(11), inp(), outp()$

Figure 11. List of states of the initial object oid_0

There is a special sequence number $i = 0$ meaning initialization of components when only δ_{int} and ta are executed. The component $D1$ fires a transition $t1$ for bindings $x = 10$, switches to a state s_1, and its ta is 20, because the fired transition $@t1(x = 10)$ holds for 20 time units. The component $D2$ sets its internal variable $x = \infty$, stays in a state s_0, and produces $ta = \infty$ because there is no action to be done.

i	τ_D	s	$\frac{\lambda}{\delta_{ext}}$	δ_{int}	$event$	ta
0	0	s_0	$\frac{none}{none}$	s_1	$t1(x = 10)$	20
1	20	s_1	$\frac{none}{none}$	s_2	$@t1(x = 10)$	0
2	20	s_2	$\frac{Y(outp)\leftarrow 10}{none}$	s_3	$-$	∞
5	20	s_3	$\frac{none}{inp\leftarrow 11}$	s_4	$-$	0
6	20	s_4	$\frac{none}{none}$	s_5	$t2(y = 11)$	∞

Figure 12. Simulation step of the component $D1$

The next step $i = 1$ activates the component $D1$ for a time 20. It fires an event $@t1(x = 10)$, changes to a state s_2, and sets $ta = 0$ because the event put a value to the place $outp$ which is mapped to output port—the value has to be added to mapped output port in next step. Step $i = 2$ generates a value 10 in the output port $outp$, switches $D1$ to a state s_3, and sets $ta = \infty$ because there is no event.

i	τ_D	s	$\frac{\lambda}{\delta_{ext}}$	δ_{int}	ta
0	0	s_0	$\frac{none}{none}$	d_0	∞
3	20	s_0	$\frac{none}{x\leftarrow 10}$	d_1	0
4	20	s_1	$\frac{Y(outp)\leftarrow 11}{none}$	d_2	∞

Figure 13. Simulation step of the component $D2$

The value is propagated through connection $D1.outp \rightarrow D2.inp$ and the external function δ_{ext} is executed on $D2$ in step $i = 3$. It removes a value from input port inp, puts it in variable x, and sets $ta = 0$ because the component has to process this value in next step. The value is propagated through connection $D2.outp \rightarrow D1.inp$ and the external function δ_{ext} is executed on $D1$ in step $i = 5$. It removes a value from input port inp and puts it in the place inp. Step $i = 6$ carries the value from place inp to place $p1$ and sets $ta = \infty$ because there is no fireable transition (the condition $x <= 10$ in $C0.t1$ is not met).

VI. Conclusion

The paper raised a question of modeling systems in real environments and a need for incorporating executive code into system models. It also presented the solution based on a combination of Petri nets and DEVS formalisms. The presented solution makes it possible to associate the formal models described by High-level Petri nets (not just OOPN) with an executable code that is incorporated into DEVS formalism structures. DEVS formalism has the advantage of being an abstract concept that can be easily adapted to a particular environment due to the basic structures and principle of the simulator. In the future, we plan to fully incorporate DEVS formalism to the implementation of the software modeling tool.

Acknowledgment

This work has been supported by the internal BUT project FIT-S-17-4014 and The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602.

References

[1] J. Warmer and A. Kleppe, The Object Constraint Language: Getting your models ready for MDA. Longman Publishing, 2003.

[2] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of uml models: a systematic review of research and practice," Software & Systems Modeling, vol. 18, no. 3, 2018, pp. 2313–2360.

[3] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, "A framework for testing uml activities based on fuml," in Proc. of 10th Int. Workshop on Model Driven Engineering, Verification, and Validation, vol. 1069, 2013.

[4] S. Guermazi, J. Tatibouet, A. Cuccuru, E. Seidewitz, S. Dhouib, and S. Gérard, "Executable modeling with fuml and alf in papyrus: Tooling and experiments," in EXE@MoDELS, 2015.

[5] R. Manione, "A full model-based design environment for the development of cyber physical systems," Design, vol. 3, no. 1, 2019, pp. 1–30.

[6] 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.

[7] R. Kočí and V. Janoušek, "Specification of Requirements Using Unified Modeling Language and Petri Nets," International Journal on Advances in Software, vol. 10, no. 12, 2017, pp. 121–131.

[8] B. Zeigler, T. Kim, and H. Praehofer, Theory of Modeling and Simulation. Academic Press, Inc., London, 2000.

[9] D. Cetinkaya, A. V. Dai, and M. D. Seck, "Model continuity in discrete event simulation: A framework for model-driven development of simulation models," ACM Transactions on Modeling and Computer Simulation, vol. 25, no. 3, 2015, pp. 17:1–17:24.

[10] V. Janoušek and R. Kočí, "PNtalk: Concurrent Language with MOP," in Proceedings of the CS&P'2003 Workshop. Warsaw University, Warsawa, PL, 2003.

[11] A. Chow and B. Zeigler, "Parallel devs: a parallel, hierarchical, modular, modeling formalism," in Proceedings of the 30th conference on Winter simulation, 1994, pp. 716–722.

[12] F. Barros, B. Zeigler, and P. Fishwick, "Multimodels and dynamic structure models: An integration of dsde/devs and oopm," in Proceedings of the 30th conference on Winter simulation, 1998, pp. 413–420.

[13] B. Zeigler, "Embedding dev&dess in devs: Characteristic behavior of hybrid models," in DEVS Integrative M&S Symposium, 2006.

[14] H. Alla and R. David, Discrete, Continuous, and Hybrid Petri Nets. Springer, 2005.