Possibilities of the Reverse Run of Software Systems Modeled by 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*—Application run tracing and step-by-step execution are an integral part of the debugging process. In many cases, debugging would be more comfortable and faster if it was possible to go back in the execution and examine the system state before getting into the wrong or disabled state. Currently, such a technique is not widespread, but there are experimental implementations that burden the application run with logging information needed to restore previous states. Moreover, many of them increase overhead in a significant way. This paper focuses on the possibility of reversing the run of systems whose behavior is described by Petri nets. The work follows the methodology of designing and validating system requirements using functional models that combine formal notation with objects of the production environment and can be used as a full-fledged application. Due to the nature of Petri Nets formalisms, it is possible to define reverse operations to reduce the overhead of application run.**

Keywords–Object Oriented Petri Nets; debugging; tracing; reverse debugging; requirements validation.

I. INTRODUCTION

This work builds on the concepts of formal approach to design and develop system requirements and evolves principles discussed in the paper [1]. The presented approach uses the formalism of Petri Nets to specify the system under development [2]. It is part of the *Simulation Driven Development* (SDD) approach [3] combining basic models of the most used modeling language Unified Modeling Language (UML) [4][5] and the formalism of Object-Oriented Petri Nets (OOPN) [6]. This approach is based on ideas of model-driven development dealing with gaps between different development stages and focuses on the usage of conceptual models during the development process of simulation models. These techniques are called *model continuity* [7]. The model continuity concept works with simulation models during design stages, while the approach based on Petri Nets focuses on *live models* that can be used in the deployed system.

When testing models or implementations, developers often use the interactive debugging technique, which allows them to go through the system run and investigate its state step by step. The logging technique and subsequent analysis of the running system are less often used. These techniques are linked to the limits of their use, notably the inability to make reverse steps. In this case, it is difficult to determine the system states before stopping (e.g., at breakpoints). However, the introduction of reverse interactive debugging leads to increased overhead, especially for running an application where it is necessary to collect the information needed to reconstruct the previous states. There are several approaches; however, they differ in their possibilities and overhead. A significant factor is, in addition to higher demands on the runtime of the application, that there is a higher demand for memory that keeps the collected information. Another issue is the overhead of reverse debugging, which is not as important as the run overhead.

There are three basic approaches to solving this problem. The first one records the system run and then performs all the steps from the beginning to the desired point (*record-replay* approach). The second approach records all the information needed to return to the previous step (*trace-based* approach). The third approach records only selected checkpoints, so they are reliably replicated (*reconstruction-based* approach). Reverse debugging is done by reconstructing the appropriate checkpoint state and then making forward steps. The approach presented in this paper is based on the trace-based reverse debugging. Due to the nature of used OOPN formalism, which has a formal base working with uniquely defined events, there is no problem in defining and performing reverse operations associated with each event.

The paper is organized as follows. Section II introduces related work. Section III summarizes basic definitions of OOPN formalism needed to define tracing concepts. Section IV discusses the possibilities of OOPN models simulation tracing and introduces the simple demonstrating model. Section V focuses on recording states and event during the simulation and Section VI describes reverse events and operations when reverse debugging performed. The question of saving the whole state is discussed in Section VII. The summary and future work are described in Section VIII.

II. RELATED WORK

The solution based on recording simulation run and replaying it from the beginning to the breakpoint may be time-consuming and, for a long run of the application, unsuitable due to time lags when debugging. As examples, we can mention Instant Replay debugger [8] or Microsoft Visual Studio 2010 IntelliTrace [9].

The trace-based solution logs all steps, so it is possible to determine the current state and the sequence of steps that led to this state. In many cases, the simulators record everything and, therefore, it is possible to go back to one of the previous

steps. The scope of that solution is limited by what and how can be traced, especially using multi-processors is very difficult to work. As examples, we can mention Green Hills Time Machine [10], Omniscient Debugger [11] fo Java Virtual Machine, or gnu reverse debugger gdb 7.0 [12]. The last mentioned, gdb debugger, is very slow but is the only open-source solution. There are tools based on Petri nets that allow reverse debugging, e.g., the TIme petri Net Analyzer TINA [13]. Nevertheless, these tools focus on a specific variant of Petri nets that are not usable for the application environment. Besides, there are also tools suitable for these purposes, e.g., Renew [14], which are similar to the SDD approach but do not allow reverse debugging.

Some solutions allow us to go back within the operation stack, change the current state, and proceed from this step. An example may be the Smalltalk language [15]. Even in this case, however, we do not have a stack of states associated with the appropriate steps. We only get the current state, which is reflected in all previously executed steps.

III. Basic definition of OOPN formalism

In this section, we introduce the basic definition of Object-oriented Petri Nets (OOPN) formalism necessary for the presented purpose.

A. System of Classes and Objects

For this work, we define the Object Oriented Petri Nets (OOPN) as a system of classes and objects that consists of the individual elements [16].

Definition 1: System OOPN is $\Pi = (\Sigma, \Gamma, c_0, o_0)$, where Σ is a system of classes, Γ is a system of objects, c_0 is an initial class and o_0 is an identifier of the initial object instantiated from the class c_0.

Definition 2: System of classes Σ consists of sets of elements constituting classes and is defined as $\Sigma = (C_\Sigma, MSG, N_O, N_M, SP, NP, P, T, CONST, VAR)$, where C_Σ is a set of classes, MSG is a set of messages, N_O is a set of object nets, N_M is a set of method nets, SP is a set of synchronous ports, NP is a set of negative predicates, P is a set of places, T is a set of transitions, CONST is a set of constants and VAR is a set of variables. Messages MSG correspond to method nets, synchronous ports, and negative predicates.

Definition 3: The class is a structure

$$C = (MSG^C, n_O^C, N_M^C, SP^C, NP^C),$$

where $MSG^C \subseteq MSG$, $n_O^C \in N_O$, $N_M^C \subseteq N_M$, $SP^C \subseteq SP$ and $NP^C \subseteq NP$. Each net of the set $\{n_O^C\} \cup N_M^C$ consists of places and transitions (subsets of P and T).

Definition 4: System of objects Γ is a structure containing sets of elements constituting the model runs (the model run corresponds to the simulation, so that we will use the notation of simulation). $\Gamma = (O_\Gamma, N_\Gamma, M_N, M_T)$, where O_Γ is a set of object identifiers, N_Γ is a set of method nets identifiers, $M_N \subset (O_\Gamma \cup N_\Gamma) \times P \times U^M$ is place markings and $M_T \subset (O_\Gamma \cup N_\Gamma) \times T \times \mathcal{P}(BIND)$ is transition markings.

Definition 5: The OOPN system universe U is defined $U = \{(cnst, cls, oid) \mid cnst \in CONST \wedge cls \in C_\Sigma \wedge oid \in O_\Gamma\}$. The system universe represents a set of all possible values that may be part of markings or variables. $U^{\mathcal{M}}$ is then a multi-set over the set U.

We can use the following notation to simplify writing. For constants, we write down their values directly, e.g., $10, 'a'$. For classes, we write down their names directly without quotes or apostrophes. To identify an object, we write its identifier with a @ character.

Definition 6: The set of all variable bindings BIND used in OOPN is defined $BIND = \{b \mid b : VAR \longrightarrow U\}$.

Definition 7: We define operators for instantiating classes Π_C and method nets Π_N that create the appropriate instances and assign them identifiers from sets O_Γ, resp. N_Γ. When creating a new instance of the class $c \in C_\Sigma$, we will write $\Pi_C(c) = o$ or $\Pi_C(c, o)$, where $o \in O_\Gamma$. Similarly, for the method net instance $m \in N_M$, we will write $\Pi_N(o, m) = n$ or $\Pi_N(o, m, n)$, where $o \in O_\Gamma$ is an object where the method net instance $n \in N_\Gamma$ is created.

Individual class elements are identified by their fully qualified names consisting of sub-element names separated by a dot. The class is identified by its name, e.g., C. The method is identified by class and method names, e.g., C.M, the method place C.M.P, and so on. In the case of object net, the elements will be written directly without method identification, e.g., C.P. Similarly, we introduce the identification of Γ object system elements. Objects and nets instances are uniquely identified by their identifiers, net elements (transitions and places) by their names. For instance, the transition $t \in T$ of the method net $m_i \in N_\Gamma$ can be identified by following notations: $m_i.t$ or (m_i, t). The object net describes the autonomous activities of the object; its instance is always created with the instantiation of the class, and is just one. For this reason, the notation $o \in O_\Gamma$ can identify the class instance as well as its object net. Method nets describe the object's response to the sent message. In case the message is received, the instance $n \in N_\Gamma$ of the respective net N_M is created, and its simulation starts.

B. OOPN Model Example

An example illustrating the basic elements of OOPN formalism is shown in Figure 1. There are defined two classes C0 and C1, where the class C0 is marked as the initial one. When you run the simulation, an instance of the initial class is created, that is, the class C0. The class C0 has only an object net that contains places $p1, p2, p3, p4$ and transitions $t1, t2, t3$. These transitions gradually create instances of the class C1 in their actions and send doFor:20 and doFor:10 messages to these objects. The object net C1 contains the place p11 with an initial marking 10 and the method net doFor:. The method net has a place x corresponding to the x argument of the method, an output place return, where the resulting method token is placed, and two transitions t1 and t2.

Transitions are protected by guards. The transition t1 is fireable (can be executed, i.e., fired) if the value of the passed argument x is less than or equal to the value stored in the place p11 of the object net (x <= s). The transition t2 is

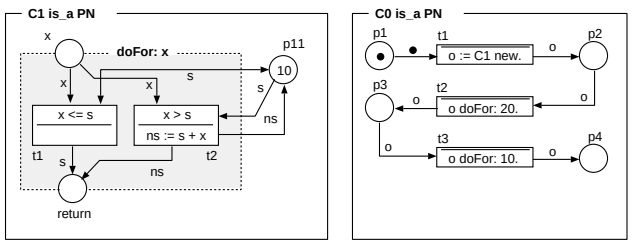

Figure 1. An example of the OOPN model with basic elements.

fireable if the value of the passed argument x is greater than the value stored in the place p11 of the object net (x > s). Firing the transition t1 does not change the state of the object, and a copy of the value from the place p11 is put to the output place return. It indicates that the method has reached a result, and its net terminates when the calling transition obtains this result. By firing the transition t2, the value of the variable x is added to the value stored in place p11.

C. Nets of the OOPN Formalism

The OOPN class consists of two categories of nets, *object net* and *method net*. These nets are instantiated, i.e., their copy is created over which a simulation is performed. When the new object is being created, its object net is instantiated immediately. When the method is invoked, its net is instantiated. Now let us take a look at the example in Figure 1 and explain the model dynamics, i.e., the use of individual nets.

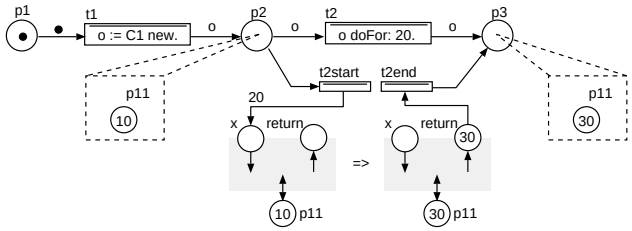

Figure 2. A usage of the method net doFor: of the object net C0.

As soon as the simulation starts, an instance of the initial class (i.e., the class C0) is created. Part of the object net of the initial object o_0 is shown in Figure 2. First, the transition t1 is performed, which instantiates the class C1 class and puts a newly created object to the place p2. In place p2, there is now an object o_1 with an object net containing only one place p11 having the initial marking 10. Next, the transition t2 is fired, which sends a message doFor: with the argument value 20 taken from the place p2. We can imagine this situation as replacing the transition t2 with two auxiliary transitions t2start, respectively t2end. These transitions are associated with input, respectively output, places of the method net its instance was created. The object o_1 now contains an object net (constituted by the place p11) and an instance of the method net doFor:. When the method net completes the execution, the resulting token is placed to the place return. The auxiliary transition t2end fetches this token from the place return and ends the method net's run. An object o_1 is inserted to the place p3, which now contains only the object net with the place p11 containing the value of 30.

D. Place

The place is represented by a named multi-set. The multi-set A^M is a generalization of the set A such that it can contain multiple occurrences of elements. Thus, the multi-set can be defined as a function $A^M : A \rightarrow \mathbb{N}$, which assigns to each element $a \in A$ the number of occurrences in the multi-set. The number of occurrences will be denoted by the term *frequency*. We denote $|A|$ the cardinality of the set A, i.e., the number of elements in the set A. We denote $|A^M|$ the cardinality of multi-set A^M, i.e., the sum of frequencies of all elements in the set A. For an individual element x of the place $p \in P$, we write $x \in p$ a for its frequency $m`x$.

Definition 8: The place marking corresponds to its content and is defined as a multi-set $M_P = \{(m, o) \mid m \in \mathbb{N}^+ \wedge o \in U\}$, where m is frequency of the member o in the multi-set. Members of multi-set will be written in the form $m`o$, marking of the place $p \in P$ will be written in the form $M_P(p) = \{m_1`o_1, m_2`o_2, \ldots\}$.

As an example we can mention the initial marking of the place $o_0.p1$ from the model shown in Figure 1— $M_P(o_0.p1) = \{1`\bullet\}$. The place cardinality is $|o_0.m_1.p2| = 1$ a $|o_0.m_1.p2^M| = 1$.

E. Transition

The transition is a net element whose execution determines the model's dynamics. OOPN dynamics are based on High-level Petri nets, but the semantics of transitions is modified. Each transition includes edges that connect the transition to places; the edge defines one condition for the fireability of the transition. The condition is defined by the inscription language. We distinguish input conditions (*precondition*), test conditions (*condition*) and output conditions (*postcondition*). The transition is fireable for certain *binding* of variables that are defined in the input and test conditions and the transition guard, if there are a sufficient number of objects in input places and the guard is evaluated truthfully for the obtained binding.

Individual types of transitions are defined as follows

- *Input condition* (*precondition*) associates a transition with a place whose state is a condition for firing (performing) the transition. When transition fires, the associated objects are removed from the associated input place. An example is the edge from place C0.p1 to the transition C0.t1.
- *Test condition* (*condition*) is similar to the input condition, except it does not change the state of the associated input place when the transition fires. The demonstration example does not include a test edge.
- *Output condition* (*postcondition*) specifies what objects are inserted in the associated output place after the transition fires. An example is the edge from the transition C0.t1 to the place C0.p2.

F. Inscription Language and Arc Expressions

An important part of OOPN formalism is its language of instruction, so-called *inscription language*. The language includes edge evaluation and operations defined in guards

and actions of transitions. The form of the script language is inspired by the Smalltalk language.

Arc expression matches the usual approach used in Petri nets. Each arc expression has a form of m‘o, where $m \in \mathbb{N}^+ \cup VAR$ and $o \in U \cup VAR$. The expression element m represents the frequency of o in the multi-set and can be denoted by a numeric value or a variable. If the variable is used at the position m, the frequency of the member o in multi-set is assigned to that variable. The element o represents the object stored in multi-set and can be defined by the element of the universe U or the variable. If a variable is used at the position o, an object from multi-set, whose frequency corresponds to specified m, is bound to that variable. If both parts of an expression are defined by variables, any object and its frequency are bound to these variables. If the content of the multi-set does not match the given expression, the bounding process fails.

G. Synchronization Mechanisms

Synchronization elements and dynamics of the OOPN model are presented in the modified model, which is shown in Figure 3. Two classes C0 and C1 are defined here. Object net of the class C0 contains C0.p1 and C0.p2 and one transition C0.t1. The object net C1 is empty. The class C0 contains the method C1.init:, the synchronous port C0.get: and the negative predicate C0.empty. The class C1 contains the method C1.doFor:.

We show an example of calling synchronous port C0.get: with a free parameter. The synchronous port has one parameter named o. This port is called from the transition C1.t2 with free variable n. The parameter o is free, too, and is bound to one object from the place C0.p2. The bound object is available in the calling transition via the variable n. If the called synchronous port does not exist, the transition is not fireable.

An example of a negative predicate is C0.empty. The predicate is called from the transition C1.t3, which means that the transition C1.t3 is fireable if the place C0.p2 is empty. Negative predicate C0.empty is complementary to the synchronous port C0.get: called with a free parameter. Either one or the other is fireable, never both or none at the same time. In this way, it is possible to model a decision based on the existence of tokens in places.

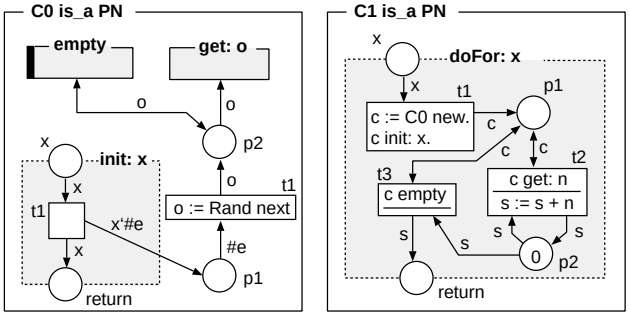

Figure 3. OOPN model with synchronization mechanisms.

Now let us look at the dynamics of the model that is created by instantiating the class C1, $\Pi_C(C1) = o_0$, and calling the method doFor: 3 on the object o_0. When the method net is being instantiated, $\Pi_N(o_0, C1.doFor:) = m_1$, and a constant object 3 is inserted into the parameter place $o_0.m_1.x$ the transition $o_0.m_1.t1$ fires. Executing this creates an instance of the class C0, $\Pi_C(C0) = o_1$, a reference to object o_1 is stored in the variable c and object o_1 is initialized by calling the method o_1.init, $\Pi_N(o_1, C0.init) = m_2$. It inserts three symbols #e to the place o_1.p1. The transition o_1.t1 generates three random numbers and puts them in the place o_1.p2. Transitions $o_0.m_1.t2$ and $o_0.m_1.t3$ are complementary. The transition $o_0.m_1.t2$ is fireable if there is at least one object (number) in the place o_1.p2. Test of fireability and obtaining objects at the same time, if the test is successful, are provided by the synchronous port o_1.get: with the free variable n. When the transition is performed, one object from the place o_1.p1 is bound to the variable n. At the same time, the bound port get: is executed on the object o_1 and the bound object is removed from the place o_1.p2. Transition $o_0.m_1.t2$ then adds the acquired value to the amount stored in the place $o_0.m_1.p2$. Figure 4 shows the principle of dynamic fusion of transition and synchronous port. Figure 4a) is the state after execution the transition $o_0.m_1.t1$. The place $o_0.m_1.p1$ contains a reference to the object o_1 and the place $o_1.p2$ contains three randomly generated values. Dynamic fusion of the transition $o_0.m_1.t2$ with synchronous port is shown in Figure 4b). After the transition $o_0.m_1.t2$ fires for the binding $xn = 10$, the model moves to the state shown in Figure 4c).

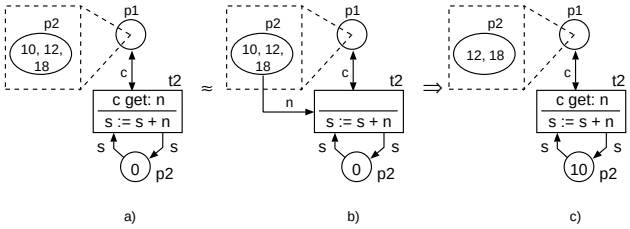

Figure 4. Dynamic fusion of the transition C1.doFor:.t2 and synchronous port C0.get:.

The transition $o_0.m_1.t3$ is fireable if the place $o_1.p2$ is empty. The fireability test is provided by calling the negative predicate o_1.empty. If the transition $o_0.m_1.t3$ fires, the value is transferred from the place $o_0.m_1.p2$ to the place $o_0.m_1.return$ and the method execution is terminated. Calling and executing the method o_1.doFor: x generates x random numbers and returns the sum of them.

H. Set of Classes

The formalism of OOPN works, in addition to the OOPN objects (O_Γ and the corresponding set of classes C_Σ), with objects that are not a direct part of the formalism. The principle of their usage is based on Smalltalk, which is also used as the inscription language of the formalism of OOPN. These objects are especially *basic constant objects* (sometimes also called *primitive objects*) such as numbers, symbols, characters, and strings. The corresponding classes will be denoted Number, Symbol, Character and String and their set, in sum, $C_\mathcal{C}$. Objects of these classes are part of the set of constants CONST. In addition to these basic objects, OOPN formalism

can work with other objects and classes. In particular, it covers collections, graphical user interface objects, user-defined classes, etc. We will call the set of these classes as *domain classes* and denote with the symbol $C_{\mathcal{D}}$, $C_{\mathcal{C}} \subset C_{\mathcal{D}}$. A set of object identifiers created from classes $C_{\mathcal{D}}$ is denoted $O_{\mathcal{D}}$.

Definition 9: Let $C_\Pi = C_\Sigma \cup C_{\mathcal{D}}$ be a set of all classes that can be used by the formalism of OOPN. Let $O_\Pi = O_\Gamma \cup O_{\mathcal{D}}$ be a set of all object identifiers that can be instantiated (created) from classes C_Π.

Definition 10: Extended Universe U_Π of OOPN is defined $U_\Pi = \{(\mathrm{cnst}, \mathrm{cls}, \mathrm{oid}) \mid \mathrm{cnst} \in \mathrm{CONST} \wedge \mathrm{cls} \in C_\Pi \wedge \mathrm{oid} \in O_\Pi\}$.

Definition 11: The set of selectors MSG can be defined as the union of the following sets: $\mathrm{MSG} = \mathrm{MSG_M} \cup \mathrm{MSG_S} \cup \mathrm{MSG_P} \cup \mathrm{MSG_D}$, where $\mathrm{MSG_M}$ corresponds to method nets, $\mathrm{MSG_S}$ corresponds to synchronous ports, $\mathrm{MSG_P}$ corresponds to predicates and $\mathrm{MSG_D}$ corresponds to domain class operations.

IV. Simulation Tracing

In this section, we briefly outline the sample model and discuss the possibilities of tracing the run of the software system described by the formalism of OOPN. We call that run *simulation*.

A. Sample Model

The basic concept will be outlined using a simple example. Figure 5 shows classes of that example. Figure 5a) depicts the initial class A1 with its object net and Figure 5b) depicts the class A2 having the only method calc: with one parameter x.

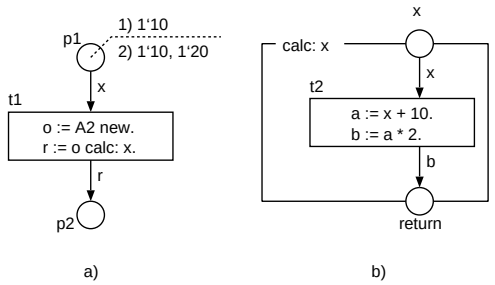

Figure 5. The sample model consisting of two classes A1 and A2; a) class A1 has only object net and b) class A2 has only method net calc:.

At the simulation start, an instance o_0 from the initial class A1 is created, $\mathrm{II_C}(\mathrm{A1}, o_0)$. The object net o_0 creates an instance of the class A2, $\mathrm{II_C}(\mathrm{A2}, o_1)$, and calls its method net calc: from the transition $o_0.t1$, $\mathrm{II_N}(o_1, \mathrm{calc:}, n_1)$. The method net n_1 executes the transition $n_1.t2$. This example works with two variants of initial marking of the object net A1; 1) $\mathrm{M_P}(p1) = \{1\text{'}10\}$ and 2) $\mathrm{M_P}(p1) = \{1\text{'}10, 1\text{'}20\}$.

B. Tracing Tree

The simulation progress can be recorded as a tree, where nodes represent the relevant unit of simulation run, and edges represent a sequence of units execution, including the bindings. The relevant unit is understood as the least set of events that the tracer records. Tree root represents the input point of the calculation. If a parallel calculation occurs during the execution of the relevant unit, this unit has more successors in its tree view. The current state of the calculation is then represented by all tree leaves. In Figure 6, we can see such a tree for the model from Figure 5 for the variant of initial marking $\mathrm{M_P}(p1) = \{1\text{'}10, 1\text{'}20\}$. In this example, the relevant unit is one executed command. Edges are recorded with a full-line arrow. Nodes capture on which net and transition the command has been performed.

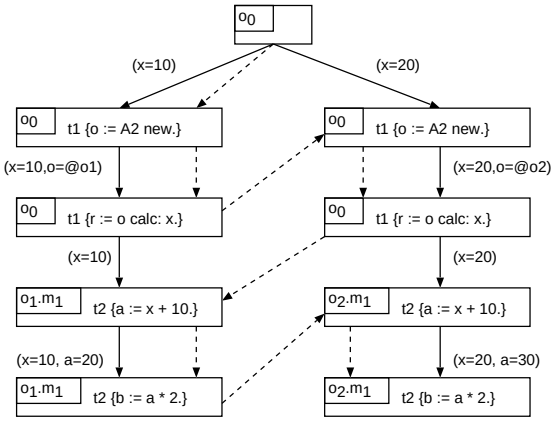

Figure 6. Scenario model of one simulation run.

The tree constructed by that way represents threads that may appear while running the simulation. It does not, however, capture the succession of steps that are important for making backward steps. The sequence of steps (events, see Section IV-C) can be different and depends on the specific conditions of the simulation run. One such variant is captured in Figure 6 with the dashed line arrows.

If all previous node states are stored during a trace, it is possible to return to any of the previous nodes at any time, make any changes to the state, and continue from that node. All nodes that are between the leaf and the selected node are removed from the tree. In this way, you can work independently on different tree branches. If there is a branch or link on the path, it must also be canceled. If another branch has been executed concurrently, it either stays (there is a connection on the way) or its execution must also be removed (there is a branch on the way).

C. Event

The simulation run is driven by events. Each executed (fired) event changes the system state, and, therefore, represents one step of model simulation. The set of states S of the system has a character of the net instances marking, which includes marking of places and transitions. One step from the state $s \in S$ to the state $s' \in S$ is written in the form $s \ [ev\rangle \ s'$, where ev is an executed event.

Definition 12: Event is $ev = (e, id, t, b)$, where e is a type of event, $id \in N_\Gamma \cup O_\Gamma$ is the identifier of net instance the event executes in, $t \in T$ is the transition to be executed (fired), and $b \in \mathcal{P}(\mathrm{BIND})$ is variables binding the event is to be executed for.

Event types can be as follows: A represents an atomic event, the entire transition is done in one step; F represents sending a message, i.e., creating an instance of a new method net and waiting for its completion; J represents completion of the method net called at F event.

(1) $\{\mathrm{m_1.p2\{1`0\}, m_1.x\{1`0\}\}}$
$\Rightarrow [(F, m_1, C1.doFor{:}.t1, \{(x, 3)\})\rangle$
(2) $\{\mathrm{m_1.p2\{1`0\}, \}}$
$\Rightarrow [(C, m_1, C1.doFor{:}.t1, \{(x, 3), (c, o_1)\})\rangle$
(3) $\{\mathrm{m_1.p2\{1`0\}, m_2.x\{1`3\}\}}$
$\Rightarrow [(A, m_2, C0.init.t1, \{(x, 3)\})\rangle$
(4) $\{\mathrm{m_1.p2\{1`0\}, m_2.return\{1`3\}, o_1.p1\{3`\#e\}\}}$
$\Rightarrow [(J, m_1, C1.doFor{:}.t1, \{(x, 3), (c, o_1)\})\rangle$
(5) $\{\mathrm{m_1.p2\{1`0\}, o_1.p1\{3`\#e\}, m_1.p1\{1`o_1\}\}}$
$\Rightarrow [(A, m_2, C0.t1, \{(\varepsilon, \#e)\})\rangle$
(6) $\{\mathrm{m_1.p2\{1`0\}, o_1.p1\{2`\#e\}, o1.p2\{1`}n_1\mathrm{\}, m_1.p1\{1`o_1\}\}}$
$\Rightarrow [(A, m_1, C1.doFor{:}.t2, \{(c, o_1), (s, 0), (n, n_1)\})\rangle$
(7) $\{\mathrm{m_1.p2\{1`}n_1\mathrm{\}, o_1.p1\{2`\#e\}, m_1.p1\{1`o_1\}\}}$
$\Rightarrow \ldots$

Figure 7. An example of the transition sequence record.

Here are a few steps of the example that is modeled in Figure 3, and its dynamics are described in Section III-G. We start from a state where $\mathrm{\Pi_N(o_0, C1.doFor{:}) = m_1}$ is invoked with the parameter 3. Places that are empty in that state are not listed.

D. Event flow subgraph

From the object point of view, we distinguish an event whose execution does not require an external stimulus (*internal event*), and an event that comes from the outside (*external event*). A feasible transition always represents the internal event. An external event can be fired by sending a message or by calling a synchronous port. The message creates a net instance consisting of an internal event sequence. Synchronous port can affect object state by modification of places content, but also invoke the sequence of internal events in the object net.

The object net can describe multiple scenarios, either interconnected or totally disjoint. The structure of each net is defined by a graph of the Petri net, so we can define the scenarios as subgraphs of such nets.

Definition 13: Let $\mathcal{S}(\mathrm{O_\Gamma \cup N_\Gamma})$ be a set of all valid subgraphs of object nets $\mathrm{O_\Gamma}$ and method nets $\mathrm{N_\Gamma}$. Individual scenarios will be denoted $\delta_\mathrm{c}(\mathrm{n}) = (\mathrm{ev_0}, \mathrm{ev_1}, \ldots)$, where $\mathrm{n} \in \mathrm{O_\Gamma} \wedge \mathrm{c} \in \mathbb{N}$.

Now, we return to the step (i.e., event) sequence entry shown in Figure 6 and write the presented scenario in the form of net subgraph,

$$\delta = ([\mathrm{A}, \mathrm{o_0}, \mathrm{t1}, (\mathrm{x} = 10)\rangle, [\mathrm{F}, \mathrm{o_0}, \mathrm{t1}, (\mathrm{x} = 10, \mathrm{o} = @\mathrm{o_1})\rangle,$$
$$[\mathrm{A}, \mathrm{o_0}, \mathrm{t1}, (\mathrm{x} = 20)\rangle, [\mathrm{F}, \mathrm{o_0}, \mathrm{t1}, (\mathrm{x} = 20, \mathrm{o} = @\mathrm{o_2})\rangle,$$
$$[\mathrm{A}, \mathrm{o_1.m_1}, \mathrm{t2}, (\mathrm{x} = 10)\rangle, [\mathrm{A}, \mathrm{o_1.m_1}, \mathrm{t2}, (\mathrm{x} = 10, \mathrm{a} = 20)\rangle,$$
$$[\mathrm{A}, \mathrm{o_2.m_1}, \mathrm{t2}, (\mathrm{x} = 20)\rangle, [\mathrm{A}, \mathrm{o_2.m_1}, \mathrm{t2}, (\mathrm{x} = 20, \mathrm{a} = 30)\rangle)$$

Now, let us go back to our example and the event $[(\mathrm{A}, \mathrm{m_1}, \mathrm{C1.doFor{:}.t2}, \{(\mathrm{c}, \mathrm{o_1}), (\mathrm{s}, 0), (\mathrm{n}, n_1)\})\rangle$. When this internal event is executed, the synchronized port $(\mathrm{o_1}, \mathrm{C0.get{:}})$ is executed simultaneously, so the net $\mathrm{o_1}$ will trigger an external event $\mathrm{C0.get{:}}$, which removes one object from the place $(\mathrm{o_1}, \mathrm{C0.p2})$. Execution of the net $\mathrm{m_1}$ corresponds to a sequence of internal events (in abbreviated notation)

$$\delta_1(\mathrm{m_1}) = ([(\mathrm{F}, \mathrm{t1})\rangle, [(\mathrm{C}, \mathrm{t1})\rangle, [(\mathrm{J}, \mathrm{t1})\rangle, [(\mathrm{A}, \mathrm{t2})\rangle,$$
$$[(\mathrm{A}, \mathrm{t2})\rangle, [(\mathrm{A}, \mathrm{t2})\rangle, [(\mathrm{A}, \mathrm{t3})\rangle)$$

and execution of the net $\mathrm{o_1}$ corresponds to the sequence of events (in abbreviated notation, external event is highlighted by superscript $^\mathrm{e}$)

$$\delta_1(\mathrm{o_1}) = ([(\mathrm{A}, \mathrm{t1})\rangle, [(\mathrm{A}, \mathrm{t1})\rangle, [(\mathrm{A}, \mathrm{t1})\rangle,$$
$$[(\mathrm{A}, \mathrm{get{:}})\rangle^\mathrm{e}, [(\mathrm{A}, \mathrm{get{:}})\rangle^\mathrm{e}, [(\mathrm{A}, \mathrm{get{:}})\rangle^\mathrm{e}.$$

For the sake of readability, it is possible not to include in the notation those components which are not important for the described situation. In our case, we only show the type of event and the transition, in other situations a simple sequence of transitions is sufficient.

E. Composite Command

If the transition contains a sequence of messages, either step-by-step or composite ones, this transition can be understood, from the OOPN theory point of view, as a sequence of simple transitions, each of which contains just one simple command. An example of such equivalence is shown in Figure 8.

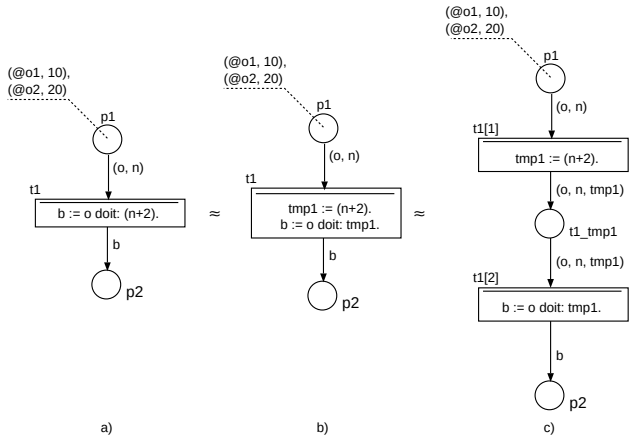

Figure 8. Composite command of the transition.

This model has four variants of execution. In the following example, only one is listed, the others are a combination of different interleaving of two concurrently running transitions

$\mathrm{t1}$. The notation of a transition using index, e.g., $\mathrm{t1}[1]$, refers to the corresponding command of the composite transition.

$$\begin{aligned}\delta_1(\mathrm{o_0}) \quad = \quad & ([\mathrm{A}, \mathrm{o_0}, \mathrm{t1}[1], \{\mathrm{o} = @\mathrm{o1}, \mathrm{n} = 10\}\rangle, \\ & [\mathrm{F}, \mathrm{o_0}, \mathrm{t1}[2], \{\mathrm{o} = @\mathrm{o1}, \mathrm{n} = 10, \mathrm{tmp1} = 12\}\rangle, \\ & [\mathrm{J}, \mathrm{o_0}, \mathrm{t1}[2], \{(\mathrm{o} = @\mathrm{o1}, \mathrm{n} = 10, \mathrm{tmp1} = 12\}\rangle, \\ & [\mathrm{A}, \mathrm{o_0}, \mathrm{t1}[1], \{(\mathrm{o} = @\mathrm{o2}, \mathrm{n} = 20\}\rangle, \\ & [\mathrm{F}, \mathrm{o_0}, \mathrm{t1}[2], \{\mathrm{o} = @\mathrm{o2}, \mathrm{n} = 20, \mathrm{tmp1} = 22\}\rangle, \\ & [\mathrm{J}, \mathrm{o_0}, \mathrm{t1}[2], \{\mathrm{o} = @\mathrm{o2}, \mathrm{n} = 20, \mathrm{tmp1} = 22\}\rangle)\end{aligned}$$

Due to that reason we slightly modify the transition marking definition, $\mathrm{M_T} \subset (\mathrm{O_\Gamma} \cup \mathrm{N_\Gamma}) \times \mathrm{T} \times \mathbb{N} \times \mathcal{P}(\mathrm{BIND})$, where $\mathbb{N}$ represents an index of the composite transition command.

V. Recording the Simulation

This section focuses on recording states during simulation. We describe each of the monitored events and how the state changes are recorded. To record the entire state would be very time consuming and memory intensive, and from the means offered by OOPN formalism point of view also unnecessary. For stepping, it is sufficient to save partial state changes that avoid storing the whole simulation image after every step.

A. State Changes Processing

A partial state change may involve inserting or selecting an element from a place, assigning the result to a variable, creating or destroying an object, creating or completing a method net instance (associated with calling and terminating this method), and creating or completing a transition instance.

1) Changing Place State: Changing the place state is the easiest operation corresponding to removing elements when transition fires or adding elements when the transition is complete. Within one step, more elements can be inserted or removed into or out of more places. The change is recorded in the following notation. We define the operation $\mathrm{add}(\mathrm{p}, \mathrm{m}, \mathrm{o})$ for adding element to the place and operation $\mathrm{del}(\mathrm{p}, \mathrm{m}, \mathrm{o})$ for removing element from the place. In both operations, $\mathrm{p} \in \mathrm{P}$ is the place, and $\mathrm{m} \in \mathbb{N}^+$ is the frequency of element $\mathrm{o} \in \mathrm{U}$.

2) Firing and Completing Composite Transition: Although the composite command in the transition is always interpreted as a sequence of individual commands, it is necessary to maintain a relationship with the original entire transition. Additionally, the transition can be run multiple times for different bindings, so it is necessary to identify the specific transition instance uniquely. Therefore, we introduce a particular event type B, which represents the transition firing for a given binding, and at the same time, assigns a unique identifier to the fired transition. Similarly, we introduce a particular C event type to completing the fired transition.

Definition 14: For writing state changes, we extend the definition of the system of objects Γ to the set of transition instance identifiers $\mathrm{T_\Gamma}$, i.e., transitions fired with a specific binding, $\Gamma = (\mathrm{O_\Gamma}, \mathrm{N_\Gamma}, \mathrm{T_\Gamma}, \mathrm{M_N}, \mathrm{M_T})$.

3) Changing Variable State: Changing the state of the variable when executing the transition is denoted by operation $\mathrm{swap}(\mathrm{t_i}, \mathrm{v}, \mathrm{o_{new}}, \mathrm{o_{old}})$, where $\mathrm{t_i} \in \mathrm{T_\Gamma}$ is a transition, $\mathrm{v} \in \mathrm{VAR}$ is the transition variable, $\mathrm{o_{new}} \in \mathrm{U}$ is a universe object assigned to the variable v and $\mathrm{o_{old}} \in \mathrm{U}$ is the original object assigned to the variable v before this event occurs.

4) Creating Object: Creating an object (a class instance) corresponds to the creation of an object net and its initialization. In terms of state recording, it is essential to keep information about the identification of newly created object $\mathrm{II_C}$ and changes of the object net's places, i.e., adding objects into places during the net initialization process.

5) Creating Method Net Instances: Creating a method net instance corresponds to a method invoking by sending a message. As with the object, it is necessary to keep information about the identification of newly created instance $\mathrm{II_N}$ and inserting objects (values) into the net's parameter places.

6) Completing Method Net Instances: After the method net instance is completed, two possible options can be applied to record changes. First, the current state of the entire net is recorded, i.e., marking of all places and all fired transitions (instances). Second, no state is recorded. The first option is more demanding for time and memory space during simulation. On the other side, it is not necessary to reconstruct the net's state so that it matches the state before its completion. The second option is more efficient during simulation, but it is more demanding to reconstruct the net's state during backward stepping. At this point, we focus on the option without state recording. We introduce a special operation $\Delta_\mathrm{N}(\mathrm{m_i})$, which indicates the completion and cancellation of the net instance $\mathrm{m_i}$.

B. Example of Tracing Simulation

We demonstrate the concept of simulation tracking on the model shown in Figure 5 for variant 1, i.e., with the initial marking $\mathrm{M_P}(\mathrm{p1}) = \{1`10\}$. For the reasons given in Section V-A2, we will modify the event definition as follows:

Definition 15: Event is $\mathrm{ev} = (\mathrm{e}, \mathrm{id}, \mathrm{t}, \mathrm{t_i}, \mathrm{b})$, where e is a kind of event, $\mathrm{id} \in \mathrm{N_\Gamma} \cup \mathrm{O_\Gamma}$ is the identifier of the net instance the event executes in, $\mathrm{t} \in \mathrm{T}$ is the transition to be executed (fired), and $\mathrm{b} \in \mathcal{P}(\mathrm{BIND})$ is variables binding for which this event is executed, and $\mathrm{t_i} \in \mathrm{T_\Gamma}$ is the identifier of fired transition.

There is only one variant of execution. The sequence of events is captured in Figure 9. For this text, we simplify writing so that we do not specify the binding b, and where it is evident from the context, we will not even specify the net identifier id.

$$\begin{aligned}\delta(\mathrm{o_1}) = ([\mathrm{B}, \mathrm{t1}, \mathrm{t1_1}\rangle, [\mathrm{A}, \mathrm{t1}[1], \mathrm{t1_1}\rangle, [\mathrm{F}, \mathrm{t1}[2], \mathrm{t1_1}\rangle, \delta(\mathrm{m_1}), \\ [\mathrm{J}, \mathrm{t1}[2], \mathrm{t1_1}\rangle, [\mathrm{C}, \mathrm{t1}, \mathrm{t1_1}\rangle) \\ \delta(\mathrm{m_1}) = ([\mathrm{B}, \mathrm{t2}, \mathrm{t2_1}\rangle, [\mathrm{A}, \mathrm{t2}[1], \mathrm{t2_1}\rangle, [\mathrm{A}, \mathrm{t2}[2], \mathrm{t2_1}\rangle, \\ [\mathrm{J}, \mathrm{t1}, \mathrm{t1_1}\rangle)\end{aligned}$$

Figure 9. Scenario (event flow) of the sample example.

The sequence of fired transitions does not necessarily correspond to the tracing tree, which also takes into account the simulation branching. The sequence of fired transitions captures a specific sequence of events, which is always unambiguously given. Figure 10 captures the sequence of events (scenario) completed with state change operations. It represents

Event	*State*
	$\Pi_{\mathrm{C}}(\mathrm{A1}, \mathrm{o}_1)$
	$\mathrm{add}(\mathrm{o}_1.\mathrm{p1}, 1, (10, \varepsilon, \varepsilon))$
$[\mathrm{B}, \mathrm{o}_1, \mathrm{t1}, \mathrm{t1}_1\rangle$	$\mathrm{del}(\mathrm{o}_1.\mathrm{p1}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{swap}(\mathrm{t1}_1, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o}_1, \mathrm{t1}[1], \mathrm{t1}_1\rangle$	$\Pi_{\mathrm{C}}(\mathrm{A2}, \mathrm{o}_2)$
	$\mathrm{swap}(\mathrm{t1}_1, \mathrm{o}, (\varepsilon, \varepsilon, \mathrm{o}_2), \varepsilon)$
$[\mathrm{F}, \mathrm{o}_1, \mathrm{t1}[2], \mathrm{t1}_1\rangle$	$\Pi_{\mathrm{N}}(\mathrm{o}_2, \mathrm{A2.calc:}, \mathrm{m}_1)$
	$\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$
$[\mathrm{B}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}, \mathrm{t2}_1\rangle$	$\mathrm{del}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}[1], \mathrm{t2}_1\rangle$	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{a}, (20, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}[2], \mathrm{t2}_1\rangle$	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{b}, (40, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{C}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}, \mathrm{t2}_1\rangle$	$\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{return}, 1, (40, \varepsilon, \varepsilon))$
$[\mathrm{J}, \mathrm{o}_1, \mathrm{t1}[2], \mathrm{t1}_1\rangle$	$\Delta_{\mathrm{N}}(\mathrm{m}_1)$
	$\mathrm{swap}(\mathrm{t1}_1, \mathrm{r}, (40, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{C}, \mathrm{o}_1, \mathrm{t1}_1\rangle$	$\mathrm{add}(\mathrm{o}_1.\mathrm{p2}, 1, (40, \varepsilon, \varepsilon))$

Figure 10. Scenario record.

a tracing simulation with storing relevant information for backward stepping. We can see state changes in the *State* column. For this text, we simplify writing events so that we do not specify the binding b.

VI. Reverse Debugging

In this section, we describe steps that are performed when stepping backwards.

A. State Changes Reverse Processing

There is a sequence of reverse operations for each state change that allows to return to the previous step. We explain the operations associated with each recorded event. Some of the operations are demonstrated on the discussed example, the first steps of reverse debugging are shown in Figure 11.

1) C-Event Type: The event C represents completing the transition instance $\mathrm{t_i}$. In a step back, our goal is to reconstruct this instance. It is necessary to perform the reverse operations that are associated with this event. Since these operations refer to the insertion of elements into the output places, the reverse operations remove these elements. The next step is to reconstruct the state of transition instance $\mathrm{t_i}$. We find the first entry regarding the instance $\mathrm{t_i}$, i.e., $[\mathrm{B}, \mathrm{t}, \mathrm{t_i}\rangle$, create this instance and perform all the swap operations. In our example, this would be a sequence of events $[\mathrm{B}, \mathrm{t1}, \mathrm{t1}_1\rangle$, $[\mathrm{A}, \mathrm{t1}[1], \mathrm{t1}_1\rangle$, $[\mathrm{F}, \mathrm{t1}[2], \mathrm{t1}_1\rangle$ and $[\mathrm{J}, \mathrm{t1}[2], \mathrm{t1}_1\rangle$. Event B ensures creation of the appropriate instance with the $\mathrm{t1}_1$ identifier. The associated sequence of swap operators is as follows: $\mathrm{swap}(\mathrm{t1}_1, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$, $\mathrm{swap}(\mathrm{t1}_1, \mathrm{o}, (\varepsilon, \varepsilon, \mathrm{o}_2), \varepsilon)$ and $\mathrm{swap}(\mathrm{t1}_1, \mathrm{r}, (40, \varepsilon, \varepsilon), \varepsilon)$. This way we filled all the variables with appropriate values, and we are in a state where the transition instance $\mathrm{t1}_1$ was completed. If the object, resp. its identifier, that has been destroyed (e.g., because it was removed by a garbage collector) is assigned to the variable, it is not essential at this point. The object will be reconstructed at the first access to it (state handling, work with method net, etc.).

2) J-Event Type: The event J represents completing the call of method. The reverse swap operation is executed, i.e., the value is removed from the variable and replaced with the original (previous) value. The next step is to perform a reverse operation $\bar{\Delta}_{\mathrm{N}}(\mathrm{m_i})$ to destroying the method net $\Delta_{\mathrm{N}}(\mathrm{m_i})$, i.e., creating net instance $\mathrm{m_i}$ and reconstructing its last state. Using operation $\bar{\Delta}_{\mathrm{N}}(\mathrm{m_i})$, we get a sequence of operations over the net $\mathrm{m_i}$ starting with $\Pi_{\mathrm{N}}(\mathrm{o_i}, \mathrm{class.method\_name}, \mathrm{m_i})$ operation. From this sequence, we will perform add and del operations on the net instance $\mathrm{m_i}$. In our example, it would be a sequence of operations $\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$, $\mathrm{del}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$ and $\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1, \mathrm{return}, 1, (40, \varepsilon, \varepsilon))$. As a result, we made method net in the state, where the place return contains the object representing number 40.

Step	*State*
$[\mathrm{C}, \mathrm{o}_1, \mathrm{t1}_1\rangle$	$\mathrm{del}(\mathrm{o}_1.\mathrm{p2}, 1, (40, \varepsilon, \varepsilon))$
$[\mathrm{J}, \mathrm{o}_1, \mathrm{t1}[2], \mathrm{t1}_1\rangle$	$\mathrm{swap}(\mathrm{t1}_1, \mathrm{r}, \varepsilon, (40, \varepsilon, \varepsilon))$
	$\bar{\Delta}_{\mathrm{N}}(\mathrm{m}_1) \Rightarrow$
	$\Pi_{\mathrm{N}}(\mathrm{o}_2, \mathrm{A2.calc:}, \mathrm{m}_1)$
	$\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{del}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{x}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{add}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{return}, 1, (40, \varepsilon, \varepsilon))$
$[\mathrm{C}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}, \mathrm{t2}_1\rangle$	$\mathrm{del}(\mathrm{o}_2.\mathrm{m}_1.\mathrm{return}, 1, (40, \varepsilon, \varepsilon))$
	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$
	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{a}, (20, \varepsilon, \varepsilon), \varepsilon)$
	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{b}, (40, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o}_2.\mathrm{m}_1, \mathrm{t2}[2], \mathrm{t2}_1\rangle$	$\mathrm{swap}(\mathrm{t2}_1, \mathrm{b}, \varepsilon, (40, \varepsilon, \varepsilon))$

Figure 11. Reverse scenario.

There may be still instances of transitions that are not terminated at the method net completion. It may happen, although it very probably indicates a mistake in the design, that there are still instances of transitions that are not terminated at the method net completion. These instances must also be reconstructed. From the sequence of operations $\bar{\Delta}_{\mathrm{N}}(\mathrm{m_i})$, we find such sequences that correspond to unfinished transitions starting with $[\mathrm{B}, \mathrm{t}, \mathrm{t_i}\rangle$ event, but having no event $[\mathrm{C}, \mathrm{t_i}\rangle$. For each such sequence we perform actions similarly to the backward step of $[\mathrm{C}, \mathrm{t_i}\rangle$ event.

3) F-Event Type: The event F represents the method invoking on the object. In the reverse step, the appropriate instance of method net specified in Π_{N} operator is destroyed.

4) A-Event Type: The event A represents the atomic execution of the operation. The reverse swap operation is executed, i.e., the value is removed from the variable and replaced with the original (previous) value. If the atomic operation is a creation of a class instance Π_{C}, this instance is destroyed.

5) B-Event Type: The event B represents the start of transition execution (creation of a transition instance). In the reverse step, the transition instance $\mathrm{t_i}$ is destroyed, and the add reverse operation is performed. There is no need to swap variables, as the entire fired transition is canceled.

B. Object Reconstruction

At the time of access to the object, e.g., due to the reconstruction of the method net, it may happen that the object no longer exists. The reason may be the loss of all references to this object and its removal by the garbage collector. At this point, it is necessary to create the object and reconstruct its last state. Because the object was destroyed, it means that there were no existing method nets. It is necessary to reconstruct the state of the object net, which is done in the same way as the method net reconstruction. The sequence of corresponding operations on the object net $\mathrm{o_i}$ is obtained by using $\overline{\Delta_{\mathrm{O}}}(\mathrm{o_i})$ operation, which is similar to $\overline{\Delta_{\mathrm{N}}}(\mathrm{o_i})$ operation, but the obtained sequence starts with $\Pi_{\mathrm{C}}(\mathrm{class}, \mathrm{o_i})$ operation and the class instance is created instead of method net instance.

VII. Saving the Simulation State

As already mentioned, one way of ensuring the consistency of the state in reverse debugging is by continuously saving the state. In this section, we outline the way of the object's state saving and the associated problems.

A. Simulation State

The problem associated with continually saving changes may be the time consuming to reconstruct a state that is far from the current location, i.e., the simulation state. The distance μ can be defined as the number of steps taken from the start of the simulation, or the last saved state, to the stop point of the simulation. From this perspective, it seems appropriate to provide continuous saving the entire state at specific distances (after a certain number of steps) and saving changes from that storage point. In this way, the process of reconstructing states according to debugging requirements can be accelerated. On the other hand, it increases the demands of both spatial (the need to store more data) and time during the simulation. The question arises as to the appropriate choice of distance μ, after which it is useful to save the whole state of the simulation.

Definition 16: Simulation state $\mathcal{S}$ corresponds to the system of objects Γ consisting of sets of nets identifiers, net markings, and transition markings, $\mathcal{S} = (\Gamma, \mathrm{R_T})$, where $\mathrm{R_T} \subseteq \mathrm{N_\Gamma} \times (\mathrm{T}, \mathbb{N})$ is a relation mapping fired method nets to fired transitions from which the nets have been called.

B. Domain objects

The second, more serious problem is related to objects that are not directly managed by the simulation engine. These are primarily objects of the production environment $\mathrm{O_\mathcal{D}}$, which are instances of the domain classes $\mathrm{C_\mathcal{D}}$. In this case, it is not possible to reconstruct the state as presented in the previous sections (except for the constant objects $\mathrm{O_\mathcal{C}}$).

We can divide domain objects into several categories and work with them accordingly when trying to save a simulation state:

- *General objects*. Their concrete form cannot be specified in any way and, thus, it is impossible to define some limitations on these objects. They cannot be easily managed.
- *Collection*. Collections can be understood as specific objects, and the simulator can be prepared for it; their storage and recovery can be ensured in a relatively simple manner. However, objects stored in collections fall into the general object category.
- *Interface to other systems*. These objects serve as wrappers for other objects and as such can be understood as constant objects. Therefore, these objects do not need to be stored. Again, the objects that the interface encapsulates are general objects.
- *Objects that are linked to the model* (that is, the model may not have references to them). These objects are mostly outside the simulation management and use model objects mainly to determine the state by which they set their own state. Typically, these are user interface objects. This group can be understood as a specific subset of general objects.
- *Simulated general objects*. These objects are mostly used instead of ordinary general objects to simulate their behavior and reactions. They are used when it is impossible to connect created models to the real environment, or it is not necessary.

C. General Domain Objects

General domain objects are a significant problem in providing state during the simulation. One solution is to use the resources of the production environment, i.e., the environment in which the domain objects are implemented, and to serialize and deserialize those objects. In many cases, however, this solution requires to modify domain class definitions, e.g., to use annotations associated with a suitable Java toolkit. But, such an approach is so demanding that it is unusable in practice. Another problem is the large object structures where an object referenced from the OOPN model contains references to other nested objects. Ensuring their consistency is a significant problem in the reconstruction of complex object structures.

The specific solution to this problem then depends on the environment in which the entire system for modeling and verifying requirements is implemented. The implementation in Smalltalk can use a relatively clear and straightforward means of storing the state of objects without having to interfere in any way with the existing domain object code [15]. In this case, the combination of continually saving state changes and saving the entire state after μ steps is applied. Generic domain objects can be stored with the entire structure, or the plunging depth can be limited.

It is necessary to take into account that when debugging with backward steps, it is not possible to ensure a complete reconstruction of all domain objects, but only the basic ones, as mentioned above. Additionally, some of the general domain objects are still in the last achieved state. Nevertheless, we have to realize that the system, for which the discussed backward stepping principle is intended, serves mainly for the analysis and verification of requirements, as presented in [17]. General domain objects are mainly used to link the created model to existing components for system validation under real conditions. Under these circumstances, it can be accepted if

some actions on the domain objects are not performed. It is still possible to trace back the location where the error occurred, or the wrong data was sent to the domain objects. If a simulated domain object is used for validation purposes, the situation at the state reconstruction is much simpler.

VIII. Conclusion

The paper dealt with the concept of tracing and reversing a run of software system modeled by Petri Nets, especially the formalism of Object Oriented Petri nets. While the model is running, the sequence of events and changes in states of individual elements are logged. The presented concept, combined with saving the execution state, is fully functional but has not yet taken into account all the possibilities of use. We have abstracted the possibility of having objects that have running method nets, even though the garbage collector collected them. The reason is the existence of cyclic dependencies but unavailable from the initial object. We also were only concerned with pure Petri Nets objects and passed the domain objects, e.g., collections, objects of user classes, etc. The way of storing the state of generic domain objects remains an open question. One possibility is to save these states only at predefined points and then reconstruct them on the fly - this would involve the actual steps taken during the reconstruction. In any case, each solution requires collaboration of the host environment or subordination of domain object management to the OOPN simulator.

At present, we have an experimental partial implementation of the tool supporting reverse debugging. We will complete the implementation in the future and focus on the limitation mentioned above. For more effective stepping, it may be necessary to introduce a reconstruction of the whole state at specific points (e.g., at breakpoints). To provide greater flexibility in interactive debugging, we will consider the possibility to define the granularity of the unit to be traced (complete transition, command, method) at runtime.

Acknowledgment

This work has been supported by 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] R. Kočí and V. Janoušek, “Tracing and Reversing the Run of Software Systems Implemented by Petri Nets,” in ThinkMind ICSEA 2018, The Tvelfth International Conference on Software Engineering Advances. Xpert Publishing Services, 2018.

[2] 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.

[3] 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.

[4] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

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

[6] M. Češka, V. Janoušek, and T. Vojnar, “Modelling, Prototyping, and Verifying Concurrent and Distributed Applications Using Object-Oriented Petri Nets,” Kybernetes: The International Journal of Systems and Cybernetics, vol. 2002, no. 9, 2002.

[7] 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.

[8] T. LeBlanc and J. Mellor-Crummey, “Debugging Parallel Programs with Instant Replay,” IEEE Transactions on Computers, vol. 36, no. 4, 1987, pp. 471–482.

[9] I. Huff, “IntelliTrace in Visual Studion 2010 Ultimate,” MSDN Blogs, http://blogs.msdn.com/b/ianhu/archive/2009/05/13/historical-debugging-in-visual-studio-team-system-2010.aspx, 2009.

[10] M. Lindahl, “The Device Software Engineer’s Best Friend,” in IEEE Computer, 2006.

[11] B. Lewis and M. Ducasse, “Using Events to Debug Java Programs Backwards in Time,” in Proc. of the ACM SIGPLAN 2003 Conference on Object-oriented programming, systems, languages, and applications (OOPSLA), 2003, pp. 96–97.

[12] The GNU Project Debugger, “GDB and Reverse Debugging,” GNU pages, https://www.gnu.org/software/gdb/news/reversible.html, 2009.

[13] F. V. B. Berthomieu, F. Peres, “Model-checking Bounded Prioriterized Time Petri Nets,” in Proceedings of ATVA, 2007.

[14] O. Kummer, F. Wienberg, and et al., “Renew – User Guide,” http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf, January 2016.

[15] A. GoldBerk and D. Robson, Smalltalk 80: The Language. Addison-Wesley, 1989.

[16] 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.

[17] R. Kočí and V. Janoušek, “Modeling System Requirements Using Use Cases and Petri Nets,” in ThinkMind ICSEA 2016, The Eleventh International Conference on Software Engineering Advances. Xpert Publishing Services, 2016, pp. 160–165.