Tracing and Reversing the Run of Software Systems Implemented 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 application interactive debugging are integral part of the software systems development process. In many cases, the possibility to execute reverse steps of the system run would make debugging easier and quicker due to examination of the system state before it got into the wrong or disabled state. Currently, techniques of reversing the system run are not widespread, but there are experimental implementations. Nevertheless, these solutions increase overhead of the application run due to the logging of the information needed to restore previous states. Moreover, many of them increase overhead in a significant way. This article 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 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, consequently, their implementation using Petri Nets [1]. It is part of the *Simulation Driven Development* (SDD) approach [2] combining basic models of the most used modeling language Unified Modeling Language (UML) [3][4] and the formalism of Object-Oriented Petri Nets (OOPN) [5]. 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* [6]. 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 to go through the system run and investigate its state step by step. The logging technique and subsequent analysis of the running system is 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 that differ in their possibilities and overhead. A very important factor is, in addition to higher demands on the runtime of 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 unambiguously defined events, there is no problem to define and perform reverse operations associated to 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 discuss 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 operation when reverse debugging performed. The summary and future work is described in Section VII.

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 [7] or Microsoft Visual Studio 2010 IntelliTrace [8].

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

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

work. As examples we can mention Green Hills Time Machine [9], Omniscient Debugger [10] fo Java Virtual Machine, or gnu reverse debugger gdb 7.0 [11]. 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 [12]. Nevertheless, these tools focus on a specific variant of Petri nets that are not not usable for the application environment. Besides, there are also tools suitable for these purposes, e.g., Renew [13], that are similar to the SDD approach but do not allow reverse debugging.

Some solutions allow to go back in the operation stack, change the current state and proceed from this step. An example may be the Smalltalk language [14]. Even in this case, however, we do not have the state of the system associated with the appropriate step, but only the current state whose image we see in the context of methods that were called.

III. Basic definition of OOPN formalism

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

A. System of classes and objects

For the purposes of this work, we define the Object Oriented Petri Nets (OOPN) as a system of classes and objects that consists of the individual elements [15].

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: 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 4: 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.

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 will write its identifier with a @ character.

Definition 5: The set of all variable bindings BIND used in OOPN is defined $BIND = \{b \mid b : VAR \longrightarrow U\}$.

Definition 6: We define operators for instantiating classes II_C and method nets II_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 $II_C(c) = o$ or $II_C(c, o)$, where $o \in O_\Gamma$. Similarly, for the method net instance $m \in N_M$, we will write $II_N(o, m) = n$ or $II_N(o, m, n)$, 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 will 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. 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 will denote $|A|$ the cardinality of the set A, i.e., the number of elements in the set A. We will 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 will write $x \in p$ a for its frequency $m`x$.

Definition 7: 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 $M_P(p) = \{m_1`o_1, m_2`o_2, \dots\}$.

C. Arc Expression

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 multi-set does not match the given expression, the bounding process fails.

D. 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. 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, $\mathrm{Character}$ and String and their set, in sum, $\mathrm{C}_\mathcal{C}$. 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 $\mathrm{C}_\mathcal{D}$, $\mathrm{C}_\mathcal{C} \subset \mathrm{C}_\mathcal{D}$. A set of object identifiers created from classes $\mathrm{C}_\mathcal{D}$ is denoted $\mathrm{O}_\mathcal{D}$.

Definition 8: Let $\mathrm{C}_\Pi = \mathrm{C}_\Sigma \cup \mathrm{C}_\mathcal{D}$ be a set of all classes that can be used by the formalism of OOPN. Let $\mathrm{O}_\Pi = \mathrm{O}_\Gamma \cup \mathrm{O}_\mathcal{D}$ be a set of all object identifiers that can be instantiated (created) from classes C_Π.

Definition 9: Extended Universe U_Π of OOPN is defined $\mathrm{U}_\Pi = \{(\mathrm{cnst}, \mathrm{cls}, \mathrm{oid}) \mid \mathrm{cnst} \in \mathrm{CONST} \land \mathrm{cls} \in \mathrm{C}_\Pi \land \mathrm{oid} \in \mathrm{O}_\Pi\}$.

IV. Simulation Tracing

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

A. Sample Model

The basic concept will be outlined using a simple example. Figure 1 shows classes of that example. Figure 1a) depicts the initial class $\mathrm{A1}$ with its object net and Figure 1b) depicts the class $\mathrm{A2}$ having the only method calc: with one parameter x.

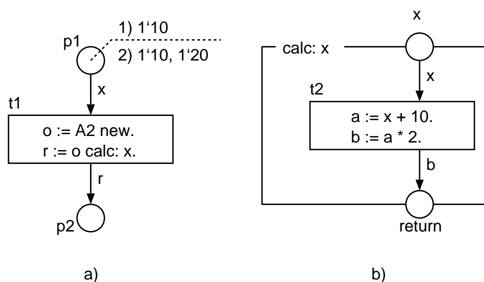

Figure 1. The sample model consisting of two classes A1 and A2; a) class A1 has only object net and b) class A2 has only method net calc:.

At the simulation start, an instance o_0 from the initial class $\mathrm{A1}$ is created, $\Pi_\mathrm{C}(\mathrm{A1}, \mathrm{o}_0)$. The object net o_0 creates an instance of the class $\mathrm{A2}$, $\Pi_\mathrm{C}(\mathrm{A2}, \mathrm{o}_1)$, and calls its method net calc: from the transition $\mathrm{o}_0.\mathrm{t1}$, $\Pi_\mathrm{N}(\mathrm{o}_1, \mathrm{calc:}, \mathrm{n}_1)$. The method net n_1 executes the transition $\mathrm{n}_1.\mathrm{t2}$. This example works with two variants of initial marking of the object net $\mathrm{A1}$; 1) $\mathrm{M}_\mathrm{P}(\mathrm{p1}) = \{1‘10\}$ and 2) $\mathrm{M}_\mathrm{P}(\mathrm{p1}) = \{1‘10, 1‘20\}$.

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 understand 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 2, we can see such a tree for the model from Figure 1 for the variant of initial marking $\mathrm{M}_\mathrm{P}(\mathrm{p1}) = \{1‘10, 1‘20\}$. In this example, the relevant unit is one executed command. Edges are recorded with a full-line arrow. Nodes capture on which network and transition the command has been performed.

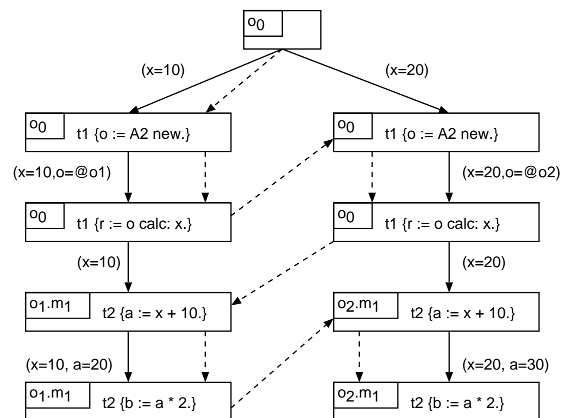

Figure 2. Scenario model of one simulation run.

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 can be different and depends on the specific conditions of the simulation run. One such variant is captured in Figure 2 with the dashed line arrows.

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 $\mathrm{s} \in \mathrm{S}$ to the state $\mathrm{s}' \in \mathrm{S}$ is written in the form $\mathrm{s}\ [\mathrm{ev}\rangle\ \mathrm{s}'$, where ev is an executed event .

Definition 10: Event is $\mathrm{ev} = (\mathrm{e}, \mathrm{id}, \mathrm{t}, \mathrm{b})$, where e is a type of event, $\mathrm{id} \in \mathrm{N}_\Gamma \cup \mathrm{O}_\Gamma$ is the identifier of net instance the event

executes in, $t \in T$ is the transition to be executed (fired), and $b \in \mathcal{P}(BIND)$ 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.

D. Event flow subgraph

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

Now, we return to the step (i.e., event) sequence entry shown in Figure 2 and write the presented scenario in the form of net subgraph, $\delta = ([A, o_0, t1, (x = 10)\rangle, [F, o_0, t1, (x = 10, o = @o_1)\rangle, [A, o_0, t1, (x = 20)\rangle, [F, o_0, t1, (x = 20, o = @o_2)\rangle, [A, o_1.m_1, t2, (x = 10)\rangle, [A, o_1.m_1, t2, (x = 10, a = 20)\rangle, [A, o_2.m_1, t2, (x = 20)\rangle, [A, o_2.m_1, t2, (x = 20, a = 30)\rangle)$.

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 3.

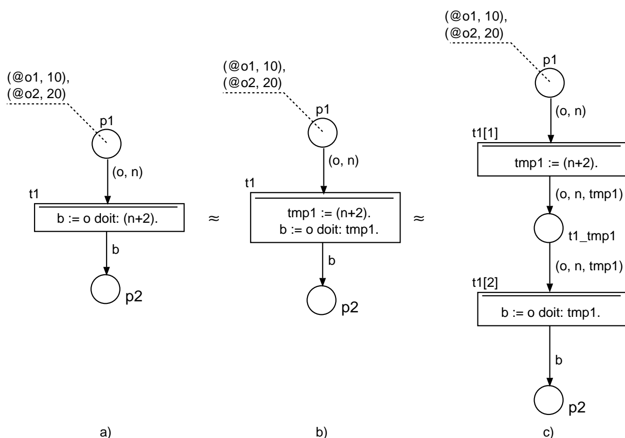

Figure 3. Composite command of the transition.

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

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

V. Recording the Simulation

This section focuses on recording states during simulation. We will 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 the purpose of stepping, it is sufficient to save partial state changes. This avoids 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 transition is complete. Within one step, more elements can be inserted or removed into or out of more places. The change will be recorded in the following notation. Operation $add(p, m, o)$ for adding element to the place and operation $del(p, m, o)$ for removing element from the place, where $p \in P$ is the place and $m \in \mathbb{N}^+$ is the frequency of element $o \in U$.

2) Firing and Completing Composite Transition: Although the composite command in the transition is always interpreted by individual commands, it is necessary to maintain a relationship to the original entire transition. Additionally, the transition can be run multiple times for different bindings, so it is necessary to uniquely identify the specific transition instance. Therefore, we will introduce a special 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 will introduce a special C event type to completing the fired transition.

Definition 12: For the purpose of writing state changes, we will extend the definition of the system of objects Γ to the set of transition instance identifiers T_Γ, i.e., transitions fired with a specific binding, $\Gamma = (O_\Gamma, N_\Gamma, T_\Gamma, M_N, M_T)$.

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

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

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 important to keep information about identification of newly created object Π_{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 identification of newly created instance Π_{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 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, but 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 will focus on the option without state recording. We will 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 1 for variant 1, i.e., with the initial marking $\mathrm{M_P(p1)} = \{1`10\}$. For the reasons given in Section V-A2, we will modify the event definition as follows:

Definition 13: 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.

The sequence of fired transitions does not necessarily correspond to the tracing tree, which also takes into account the simulation branching. Sequence of fired transitions captures a specific sequence of events, which is always unambiguously given. Figure 4 captures the sequence of events (scenario) completed with state change operations. This is about tracing a simulation with storing relevant information for backward stepping. We can see individual state changes in the *State* column. For the purpose of this text, we will simplify writing events so that we do not specify the binding b.

Event	*State*
	$\Pi_{\mathrm{C}}(\mathrm{A1}, \mathrm{o_1})$
	$\mathrm{add}(\mathrm{o_1.p1}, 1, (10, \varepsilon, \varepsilon))$
$[\mathrm{B}, \mathrm{o_1}, \mathrm{t1}, \mathrm{t1_1}\rangle$	$\mathrm{del}(\mathrm{o_1.p1}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{swap}(\mathrm{t1_1}, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o_1}, \mathrm{t1}[1], \mathrm{t1_1}\rangle$	$\Pi_{\mathrm{C}}(\mathrm{A2}, \mathrm{o_2})$
	$\mathrm{swap}(\mathrm{t1_1}, \mathrm{o}, (\varepsilon, \varepsilon, \mathrm{o_2}), \varepsilon)$
$[\mathrm{F}, \mathrm{o_1}, \mathrm{t1}[2], \mathrm{t1_1}\rangle$	$\Pi_{\mathrm{N}}(\mathrm{o_2}, \mathrm{A2.calc{:}}, \mathrm{m_1})$
	$\mathrm{add}(\mathrm{o_2.m_1.x}, 1, (10, \varepsilon, \varepsilon))$
$[\mathrm{B}, \mathrm{o_2.m_1}, \mathrm{t2}, \mathrm{t2_1}\rangle$	$\mathrm{del}(\mathrm{o_2.m_1.x}, 1, (10, \varepsilon, \varepsilon))$
	$\mathrm{swap}(\mathrm{t2_1}, \mathrm{x}, (10, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o_2.m_1}, \mathrm{t2}[1], \mathrm{t2_1}\rangle$	$\mathrm{swap}(\mathrm{t2_1}, \mathrm{a}, (20, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{A}, \mathrm{o_2.m_1}, \mathrm{t2}[2], \mathrm{t2_1}\rangle$	$\mathrm{swap}(\mathrm{t2_1}, \mathrm{b}, (40, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{C}, \mathrm{o_2.m_1}, \mathrm{t2}, \mathrm{t2_1}\rangle$	$\mathrm{add}(\mathrm{o_2.m_1.return}, 1, (40, \varepsilon, \varepsilon))$
$[\mathrm{J}, \mathrm{o_1}, \mathrm{t1}[2], \mathrm{t1_1}\rangle$	$\Delta_{\mathrm{N}}(\mathrm{m_1})$
	$\mathrm{swap}(\mathrm{t1_1}, \mathrm{r}, (40, \varepsilon, \varepsilon), \varepsilon)$
$[\mathrm{C}, \mathrm{o_1}, \mathrm{t1_1}\rangle$	$\mathrm{add}(\mathrm{o_1.p2}, 1, (40, \varepsilon, \varepsilon))$

Figure 4. Scenario record.

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 will be demonstrated on discussed example, first steps of reverse debugging are shown in Figure 5.

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 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 are therefore the removal of 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 $\overline{\Delta}_{\mathrm{N}}(\mathrm{m_i})$ 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 $\overline{\Delta}_{\mathrm{N}}(\mathrm{m_i})$, 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 $\mathrm{add}(\mathrm{o_2.m_1.x}, 1, (10, \varepsilon, \varepsilon))$, $\mathrm{del}(\mathrm{o_2.m_1.x}, 1, (10, \varepsilon, \varepsilon))$ and $\mathrm{add}(\mathrm{o_2.m_1}, \mathrm{return}, 1, (40, \varepsilon, \varepsilon))$. As a result, we made method net in the state, where the place return contains the object representing number 40.

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

Figure 5. Reverse scenario.

It may happen 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 $\bar{\Delta}_\mathrm{N}(\mathrm{m}_\mathrm{i})$, we find such sequences that correspond to unfinished transitions starting with $[\mathrm{B}, \mathrm{t}, \mathrm{t}_\mathrm{i}\rangle$ event, but having no event $[\mathrm{C}, \mathrm{t}_\mathrm{i}\rangle$. For each such sequence we perform actions similarly to the backward step of $[\mathrm{C}, \mathrm{t}_\mathrm{i}\rangle$ 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 II_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 II_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 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 o_i is obtained by using $\bar{\Delta}_\mathrm{O}(\mathrm{o}_\mathrm{i})$ operation, which is similar to $\bar{\Delta}_\mathrm{N}(\mathrm{o}_\mathrm{i})$ operation, but the obtained sequence starts with $\mathrm{II}_\mathrm{C}(\mathrm{class}, \mathrm{o}_\mathrm{i})$ operation and the class instance is created instead of method net instance.

VII. CONCLUSION

The paper dealt with the concept of tracing and reversing run of software system modeled by Petri Nets, especially the formalism of Object Oriented Petri nets. Presented concept is fully functional, but has not yet taken into account all the possibilities of use. We were only concerned with pure Petri Nets objects and passed the domain objects, e.g., collections, objects of user classes etc. We have also abstracted the possibility of having objects that have running method nets, even though they were collected by garbage collector. The reason is an existence od cyclic dependencies but unavailable from the initial object. The last constraint is to omit the method from the external (domain) object. 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 above-mentioned limitations.

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] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] M. Lindahl, “The Device Software Engineers Best Friend,” in IEEE Computer, 2006.

[10] 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.

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

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

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

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

[15] 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.