Requirements Validation Through Scenario Generation and Comparison

Radek Kočí

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

***Abstract*—When designing systems, we must solve many problems associated with the correct definition of system requirements, the right understanding, and proper implementation. Finding that design or implementation contains an error or is incomplete, and identifying where a change needs to be made, are different issues that require different approaches. Models and diagrams, often diagrams from the Unified Modeling Language (UML), are used to capture the system's requirements and basic design. The basic ones include the domain model, use case diagram, activity diagram, and scenario models. Scenarios show the communication and cooperation of objects in solving the use case under specific conditions. If the system is implemented following the design, it is possible to generate scenarios at runtime (either actual implementations or using simulation models). Thus, we can have assumed scenarios of the investigated use case's behavior and real scenarios reflecting the performed design. In many cases, it is not useful to have a detailed view of the entire communication between objects. However, it is enough to focus on specific parts, such as messages or states of objects. In this paper, we will focus on detecting discrepancies between expected and actual behavior and quickly identifying the problem's location through scenarios.**

***Keywords*–Requirements modeling; simulation; scenarios.**

I. Introduction

When designing systems, we have to solve many problems associated with the correct definition of system requirements, the right understanding, and proper implementation. There are many ways to approach these problems, but their common denominator is always verifying the correctness and correcting possible problems. Finding that design or implementation contains an error or is incomplete, and identifying where a change needs to be made, are different issues that require different approaches.

Models and diagrams, often diagrams from the UML language, are used to capture the system's requirements and basic design. The basic models include class diagrams, use case diagrams, and activity diagrams. The domain model (class diagram) depicts the basic concepts of the proposed system. The use case diagram summarizes the possibilities of using the system. The activity diagram captures the system's behavior in various conditions (it is a workflow defining individual use cases).

An integral part of the requirements and design analysis should be scenario modeling. Scenarios are an essential element, as they show the communication and cooperation of objects in solving the use case under specific conditions. Thus, one use case may have multiple scenarios, which may differ in certain parts. If the system is implemented (at least for verification purposes) following the design, it can generate scenarios at runtime (either actual implementations or using simulation models). Thus, we can have assumed scenarios of the investigated use case's behavior and current scenarios reflecting the created design. As already mentioned, one use case can have several different scenarios. However, the structure of the scenario is usually the same for the learned set of conditions. Therefore, it is possible to use scenarios to compare the expected and actual course of solving the use case.

Many tools allow you to set various conditional breakpoints and record the passage through set points. However, in many cases, it is necessary to reconstruct (or record) the entire path to the breakpoint (including information on the conditions achieved) at least from a specified point in time. A suitable means is to generate scenarios according to preset criteria. In many cases, it is not useful to have a detailed view of the entire communication between objects. Still, it is enough to focus on specific parts, messages, states of objects, etc. This paper will focus on how to detect differences between expected and actual behavior and quickly identify the problem's location through scenarios. We will focus only on selected problems of requirements and design validation through scenarios.

The paper is structure as follows. First, we introduce the basic principles of the work in Section III. The demonstration case study is described in Section IV. Then, problems of scenario modeling and validation are introduced in Sections V and VI.

II. Related Work

This work is part of the *Simulation Driven Development* (SDD) approach [1][2], which combines basic models of the most used modeling language Unified Modeling Language (UML) [3][4] and the formalism of Object-Oriented Petri Nets (OOPN) [5].

One of the fundamental problems associated with software development is the specification and validation of the system requirements [6]. The use case diagram from UML is often used for requirements specification, which is then developed by other UML diagrams [7]. The disadvantage of such an approach is an inability to validate the specification models and it is usually necessary to develop a prototype, which is no longer used after fulfilling its purpose. Utilization of OOPN formalism enables the simulation (i.e., to execute models), which allows to generate and analyze scenarios from specification models. All changes enforced during the validation process are entered directly in the specification model, which means that it is not necessary to implement or transform models.

There are methods of working with modified UML models that can be transformed to the executable form automatically. Some examples are the Model Driven Architecture (MDA) methodology [8], Executable UML (xUML) [4] language, or

Foundational Subset for xUML [9]. These approaches are faced with a problem of model transformations. It is hard to transfer back to model all changes that result from validation process and the model becomes useless. Further similar work based on ideas of model-driven development deals with gaps between different development stages and focuses on the usage of conceptual models during the simulation model development process [10]. This approach is called *model continuity*. While it works with simulation models during design stages, the approach proposed in this paper focuses on *live models* that can be used in the deployed system.

III. Initial Assumptions

In this section, we will briefly describe the initial assumptions of the work. It consists of the basis of presented concepts and the way on how we will demonstrate their usage.

A. Basic Concepts

As already mentioned, we will deal only with selected possible uses of scenarios for requirements validation. Among the most important are in particular:

- During the development of requirements, scenarios of correct behavior under the given conditions were specified. Our goal is to verify this behavior on the created model or part of the implementation. In other words, verify that the messaging sequence matches the expected behavior.
- It is necessary to find out when and under what conditions a specific method is called.
- It is needed to verify whether a specific method is always called under certain conditions.

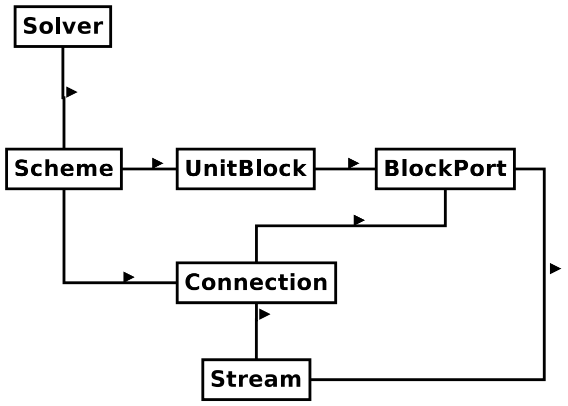

Figure 1. Domain model.

Because it is a simulation verification, it is always dependent on simulation (test) data. In our view, however, we are based on scenarios prepared in advance during the creation of requirements and design. Suppose the models are modified during the development process. In that case, these scenarios are modified (here we come across the MDE condition, namely that we always try to work at the model level).

B. Demonstration method

We will use the following procedure to demonstrate the possibilities of working with scenarios. We will present an example containing one simulation step in the balance calculation tool. We will design a domain model and a sequence diagram according to the standard procedure. We can create a workflow using Petri nets that allow us to generate scenarios. We then make a so-called scenario model based on these scenarios, which can be compared with scenarios generated under different conditions. In the next step, we will include a new request, which will be reflected in the addition of new calls to the scenarios. We modify the created scenario model and then compare it again with various generated scenarios.

IV. Case Study

In this section, we will present a simple example based on the part of the software solution of a tool for the simulation of balance calculations of technological processes. This part concerns the execution of one calculation step. We will present only the part of the calculation step that is essential for explaining the concept.

A. Domain Model

The basis of each design is a domain model that captures the basic concepts of the proposed system. These concepts, modeled mostly as classes, appear in other models describing objects' behavior or interaction. Technological processes are modeled by units (blocks) that work with input streams (e.g., water, air, gas) and generate output streams. During processing, the blocks recalculate the output streams' properties following the input streams and block settings.

```
Data:
simList : a list of blocks
forall b ∈ simList do
    initialize b
end
forall b ∈ simList do
    if b.hasChanged() then
        b.innerFunction()
        b.outFunction()
        foreach p ∈ b.ports do
            if p.hasChanged() then
                recalculate a stream
                copy a stream
                send a stream copy to the connection
            end
        end
    end
end
```

Figure 2. Description of the Balance calculation Use case.

The basic domain model is shown in Figure 1. It contains classes modeling the following concepts: blocks (`UnitBlock`), block ports (`BlockPort`), port connections (`Connection`) and streams (`Stream`). Each port stores information about the associated stream, streams are transmitted between blocks via a connection. The class `Scheme` models the schema containing blocks and joints. Balance calculations are then controlled by `Solver`.

B. Behavioral Model

A UML use case diagram is often used as the default model specifying individual use cases to capture system requirements. The behavior of use cases is then described in the text or modeled by other diagrams, such as the activity diagram. However, it is possible to use different formalisms, such as Petri nets. The chosen concept then defines in what detail the use case's behavior can be specified and how difficult it is to simulate the models created in this way due to requirements verification or transform into the selected source code. For our purposes, we will choose only one use case, namely performing a balance calculation. Its basic form is outlined in Figure 2.

Data:
b is a block
$\mathrm{p} \in \mathrm{b.ports}$ is an output port of the b
$\mathrm{s} \in \mathrm{p}$ is a steam associated to the port p
$\mathrm{p.setAttr(value)}$
$\mathrm{s.setValue(value)}$
$\mathrm{p.setChanged()}$

Figure 3. Description of the attribute changing.

Each block models different technological units, and therefore the calculations are different too. However, the basic structure is the same, and from the simulation point of view, the critical question is whether or not any of the output streams have changed. Assume that each port has a flag set when any attribute of the associated stream from the output function changes. In this case, the behavior description could look like the one shown in Figure 3.

C. Workflow Model

To model behavior as workflow, the formalism of Petri Nets can be used. The model is conceived as a sequence of events, i.e., transitions, whether internal or external. The execution of an event may be conditional, and it is possible to define different branches and, thus, different specific use case execution scenarios. An event's execution may involve sending a message to another object, or the event may be executed in response to an incoming event. In the classical concept, it is necessary to map individual sent messages to specific methods of classes, which makes it difficult to read and understand the model. When using Petri nets, the scenario is clearly defined by a sequence of events (i.e., transitions), whether internal or external.

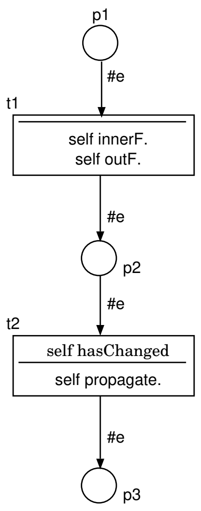

Figure 4. The *calculation* workflow.

Figure 4 shows the workflow modeling method for the Balance calculation use case from Figure 2. The workflow models the behavior for one specific calculation block. Figure 5 shows the workflow modeling method for the method *outF()*. The workflow models one possible scenario consisting of set one attribute of the port `@p` with value `10`.

V. Scenario Modeling

One scenario corresponds to a sequence of interactions between individual system objects or system objects and users. Interactions are often written in the form of a diagram, the most commonly used in this area being an activity diagram and a sequence diagram from UML. The activity diagram is suitable for modeling the whole use case's behavior, while the sequence diagram captures one specific use case scenario. This section will introduce the possibilities of using sequence diagrams as a base for scenario modeling.

A. Predefined Scenarios

Scenarios help to specify the correct, expected system behavior for a particular task. As already mentioned, scenarios are often modeled using sequence diagrams. The disadvantage is that the designer often creates these diagrams manually and must follow the rules for their creation, such as following the names defined by the domain model.

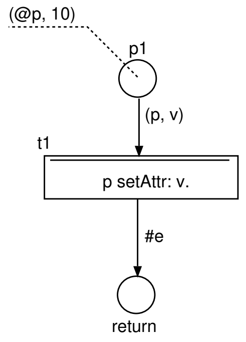

Figure 5. The *outF* workflow – one scenario.

However, if we have, in addition to the domain model, also created models of behavior as a workflow, it is possible to generate these scenarios and make our work easier. A small example of such a workflow, created using Petri nets, is shown in Figures 4 and 5. Figure 4 shows part of the method *calculate* of the *UnitBlock* concept (class), and Figure 5 shows part of the *outF* method's behavior.

The problem is that the outF method captures only one possible scenario, while the sequence diagram allows you to capture different variants of similar behavior. In this article, we will not deal with the possibilities of sequence diagrams. We will only outline this problem on a more complex diagram to capture the behavior caused by sending the method *calculate*, i.e., by performing the appropriate use case. The diagram is shown in Figure 6.

B. Scenario Definition

To define the scenario model, we start from the description of scenarios described in [11]. These scenarios work with Petri net models but can be easily adapted to messaging-defined scenarios. The scenario model is described as a messaging sequence, where messages can be grouped into blocks. These blocks represent one sub-scenario. There may be messages and sub-scenarios in the model, which may be repeated – it is possible to define their repetitions using regular expressions.

Each captured message is a pair of $\mathrm{msg} = (\mathrm{msg}^{\mathrm{s}}, \mathrm{msg}^{\mathrm{r}})$ representing the sending of the message and its return (termination). Between $\mathrm{msg}^{\mathrm{r}}$ and $\mathrm{msg}^{\mathrm{r}}$, there may be a sequence of additional messages that express the calculation to achieve the desired goal of the msg message.

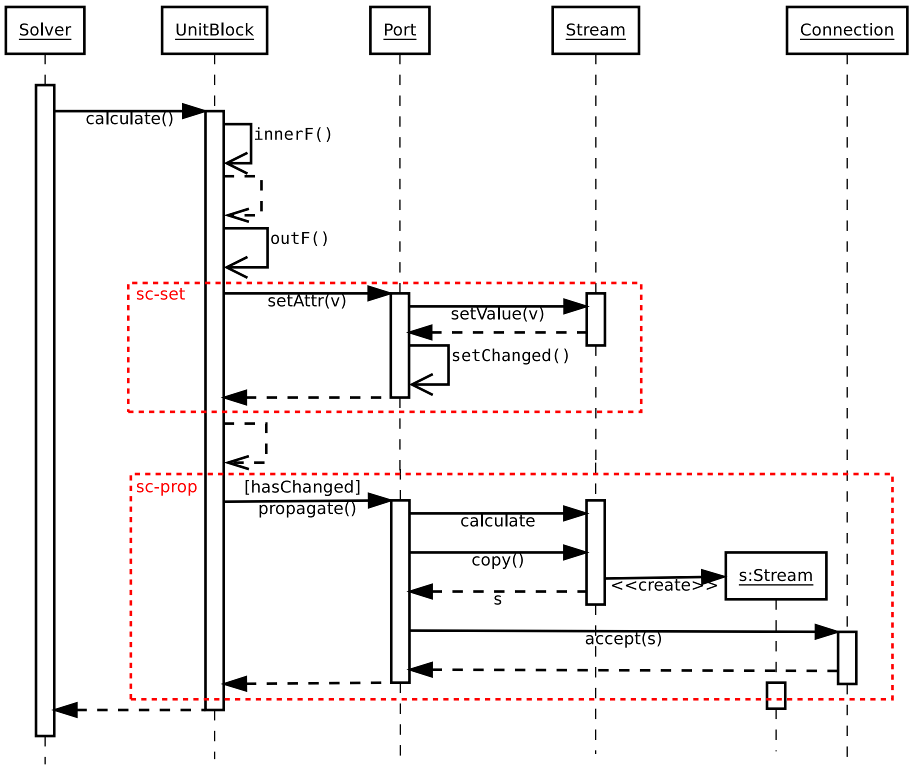

Figure 6. Sequence diagram of the balance calculation behavior.

The message $\mathrm{msg^s}$ is defined in the model as a parameterized tuple $\mathrm{msg^s = (C_1\{o_1\}, C_2\{o_2\}, msg_n\{a_1, ..., a_n\})}$, where $\mathrm{C_1}$ is the classifier of the class whose instance sends the message $\mathrm{msg_n}$ ($\mathrm{msg_n}$ is the identifier of the sent message) of the object of class $\mathrm{C_2}$. Each of the listed elements can be parameterized; the parameters are given in curly braces. For the class classifier it is possible to mark (name) their instances ($\mathrm{o_1}$, $\mathrm{o_2}$), for the sent message its attributes can be defined ($\mathrm{a_1, ..., a_n}$). Attributes can have a form of specific values or just formal parameters.

The message $\mathrm{msg^r}$ is defined in the model as a parameterized tuple $\mathrm{msg^r = (C_1\{o_1\}, C_2\{o_2\}, msg_n\{a_1, ..., a_n\}, ret)}$, where the first three elements semantically correspond to the message $\mathrm{msg^s}$ and ret is the return value (object) of the message. This value can be a specific object, variable, or special symbol ε representing the information that the method returns nothing or the return value is not important from the scenario definition point of view.

We denote the scenario model by the symbol δ. The model consists of a sequence of symbols $\mathrm{msg^s}$, $\mathrm{msg^r}$, and δ, which can be repeated according to the given rules. The rules are simple. It is necessary to follow the pairing of $\mathrm{msg^s}$ and $\mathrm{msg^r}$, and the syntax of the notation. The rules can be described by a context-free grammar $\mathrm{G_M = (\Sigma, N, P, \{S\})}$, where $\Sigma = \mathrm{\{msg^s, msg^r, \delta, *\}}$ ($*$ represents the iteration symbol, i.e., the possibility of repetition), $\mathrm{N = \{S\}}$ and P is a set of rewriting rules in the following form.

$$\begin{aligned} \mathrm{S} &\Rightarrow \delta\ \mathrm{S} \\ \mathrm{S} &\Rightarrow \delta *\ \mathrm{S} \\ \mathrm{S} &\Rightarrow \mathrm{msg^s\ S\ msg^r} \end{aligned}$$

When checking compliance with the rule, it is usually unnecessary to examine the parameters, only whether the correct syntax has been followed. This possibility can be expressed in grammar, either by engaging in the above context-free grammar or by creating a regular grammar.

The example scenario model from our example then looks like this. First, we define a sub-scenario $\delta_{\mathrm{sc\_set}}$ for setting the attributes corresponding to the red highlighted sequence *sc-set* in Figure 6.

$$\begin{aligned} \delta_{\mathrm{sc\_set}} = \ & \mathrm{(UnitBlock, Port, setAttr\{v\})}, \\ & \mathrm{(Port, Stream, setValue\{v\})}, \\ & \mathrm{(Port, Stream, setValue\{v\}}, \varepsilon), \\ & \mathrm{(Port\{p\}, Port\{p\}, setCahnged)}, \\ & \mathrm{(Port\{p\}, Port\{p\}, setCahnged}, \varepsilon), \\ & \mathrm{(UnitBlock, Port, setAttr\{v\}}, \varepsilon) \end{aligned}$$

Another sequence that can be repeated is marked in red in Figure 6. Part of this sequence is captured as a sub-scenario $\delta_{\mathrm{sc\_prop}}$. The scenario captures only significant points for the idea; the whole scenario would be unnecessarily long in this listing.

$$\begin{aligned} \delta_{\mathrm{sc\_prop}} = \ & \mathrm{(UnitBlock, Port, propagate)}, \\ & ... \\ & \mathrm{(Port, Stream, copy)}, \\ & \mathrm{(Port, Stream, copy, s)}, \\ & ... \\ & \mathrm{(Port, Connection, accept\{s\})}, \\ & ... \\ & \mathrm{(UnitBlock, Port, propagate}, \varepsilon) \end{aligned}$$

The resulting model scenario, which corresponds to Figure 6, is then captured by the scenario δ_{m}.

$$\begin{aligned}\delta_{\mathrm{m}} = \;& (\mathrm{Solver}, \mathrm{UnitBlock}, \mathrm{calculate}),\\ & (\mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{innerF}),\\ & (\mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{innerF}, \varepsilon),\\ & (\mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{outF}),\\ & \delta_{\mathrm{sc\_set}}*,\\ & (\mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{UnitBlock}\{\mathrm{b}\}, \mathrm{outF}, \varepsilon),\\ & \delta_{\mathrm{sc\_prop}}*,\\ & (\mathrm{Solver}, \mathrm{UnitBlock}, \mathrm{calculate}, \varepsilon)\end{aligned}$$

VI. Scenario Validation

The validation is then performed by comparing the model scenario with the actual scenario, respecting the regular expression's control characters. A tool based on finite state machines can be used for evaluation. Evaluation can take place in several modes, depending on the type of authentication required.

- *Entire scenario validation.* We verify the whole sequence of the scenario. If we encounter a deviation, we record an error at this point. In this way, it is possible to verify that a method should not be called; that the method is not called in the correct place; or the method with the wrong parameters (attributes, object sending the message, the object receiving the message) is called.
- *Pass validation.* The model defines only the key aspects that must be followed in that order. If there are other calls outside these defined points, they are ignored for evaluation. It can be used, for example, if we are only interested in the question of whether a particular message is sent after another message has been executed.

A. *Entire Scenario Validation*

We will now introduce these verification concepts with our examples. Let us start with the whole sequence. We must first obtain a scenario of the actual models or implementations. We modify the original workflow to new conditions and then generate different scenarios, which, however, must structurally correspond to the model scenario. Such a workflow modification is shown in Figure 7 – in this case, the attribute is set more than once.

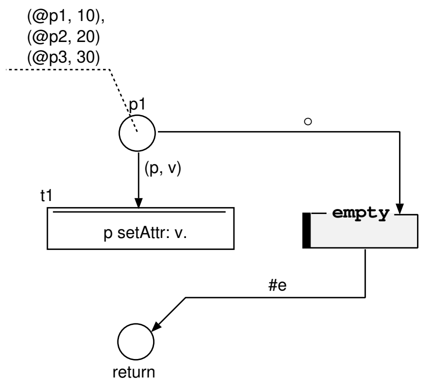

Figure 7. The *outF* workflow – second scenario.

The generated scenario δ_1 then corresponds to the original scenario from Figure 6, only the part marked *sc-set* is replaced by the sequence of calls from Figure 8. Sequence of calls $\delta_{\mathrm{sc-set1}}$ and $\delta_{\mathrm{sc-set2}}$ corresponding to marked blocks *sc-set1* and *sc-set2* in Figure 8 is as follows (only an example for *sc-set1* is presented).

$$\begin{aligned}\delta_{\mathrm{sc\_set1}} = \;& (\mathrm{UnitBlock}, \mathrm{Port}, \mathrm{setAttr}\{10\}),\\ & (\mathrm{Port}, \mathrm{Stream}, \mathrm{setValue}\{10\}),\\ & (\mathrm{Port}, \mathrm{Stream}, \mathrm{setValue}\{10\}, \varepsilon),\\ & (\mathrm{Port}\{\mathrm{p}\}, \mathrm{Port}\{\mathrm{p}\}, \mathrm{setChanged}),\\ & (\mathrm{Port}\{\mathrm{p}\}, \mathrm{Port}\{\mathrm{p}\}, \mathrm{setChanged}, \varepsilon),\\ & (\mathrm{UnitBlock}, \mathrm{Port}, \mathrm{setAttr}\{10\}, \varepsilon)\end{aligned}$$

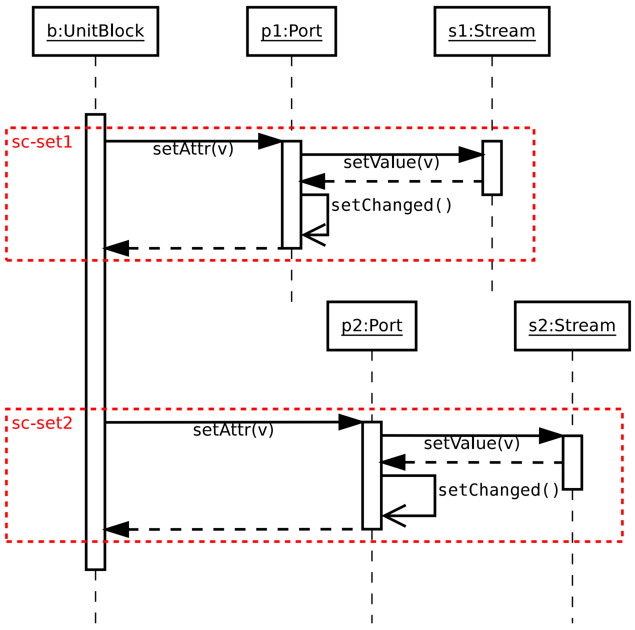

Figure 8. Sequence diagram of the extended *outF* workflow.

By comparing the sub-scenario $\delta_{\mathrm{sc\_set}}$ with the sequence of scenarios $\delta_{\mathrm{sc\_set1}}$ and $\delta_{\mathrm{sc\_set2}}$ we find that they are structurally identical, only substitutions $\{\mathrm{v}/10\}$ and $\{\mathrm{v}/20\}$ occur. Then, it can be concluded that $\delta_{\mathrm{sc_set}}* == \delta_{\mathrm{sc_set1}}, \delta_{\mathrm{sc_set2}}$ and then $\delta_{\mathrm{m}} == \delta_1$. The newly generated scenario thus corresponds to the scenario model.

B. *Pass Validation*

We will show a variant where we will not be interested in the whole scenario, but only the fulfillment of some condition. We will create/generate a model scenario containing only those calls that we consider crucial for validation. In our example, this can be the condition the propagate method must always be called after any call of the *setAttr* method. The model scenario can then look like this. The newly generated scenario thus corresponds to the scenario model.

$$\begin{aligned}\delta_{\mathrm{pass}} = \;& (\mathrm{UnitBlock}, \mathrm{Port}, \mathrm{setAttr}\{\mathrm{v}\}),\\ & (\mathrm{UnitBlock}, \mathrm{Port}, \mathrm{propagate})\end{aligned}$$

When comparing the model scenario with the generated one, we will only be interested in whether the above sequence is followed and other parts of the scenario will be uninteresting.

C. *New Functionality Validation*

The last example is the addition of new functionality to an existing requirements model and implementation. This functionality refers to a new type of attribute change propagated backward, i.e., through input streams back to input blocks. When setting the attribute, the given port must be set as changed, and at the end of the use case, the *backProp* method must be called, which will ensure data transfer in the correct direction. A possible scenario for this behavior is shown in Figure 9.

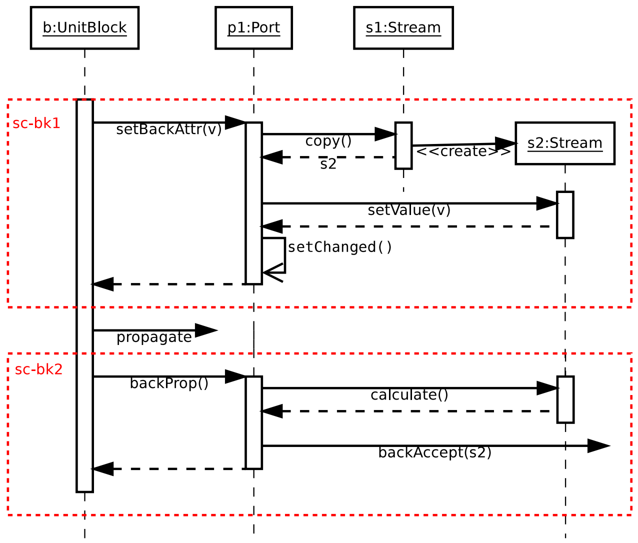

Figure 9. Scenario of the new functionality – backProp.

A model scenario δ_{back} verifying the correctness of the primary sequence of messages is shown in the following statement.

$$\begin{aligned}\delta_{\text{back}} = \;& (\text{UnitBlock}, \text{Port}, \text{setBackAttr}\{\text{v}\}),\\ & (\text{Port}, \text{Stream}, \text{copy})\\ & (\text{Port}, \text{Stream}, \text{copy}, \text{s2})\\ & (\text{UnitBlock}, \text{Port}, \text{backProp})\\ & (\text{Port}, \text{Connection}, \text{backAccept}\{\text{s2}\})\end{aligned}$$

VII. Conclusion

In this paper, we introduced the basic concept of requirements validation and its implementation through scenarios. Scenarios can be described in various ways, such as sequence diagrams. However, workflows offer a more general description ability than a sequence diagram and allow the generation of specific scenarios or models, i.e., some patterns that can then be used for comparison. The workflow can be modeled, for example, by Petri nets, as briefly shown in this paper. Real scenarios can then be obtained either by modifying the workflow or directly from the implementation if a tool was available that captures essential information for generating sequence diagrams or their parts.

We currently have a tool for generating sequence diagrams from models described by Petri nets. The presented concept works only with a structural comparison. In the future, it seems to be an interesting possibility to parameterize the sequences themselves. This feature would make it possible, for verification purposes, to specify precisely which specific objects are involved in the communication, in what state, etc.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602.

References

[1] 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.

[2] 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.

[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. 31, no. 9/10, 2002, pp. 1289–1299.

[6] K. Wiegers and J. Beatty, Software Requirements. Microsoft Press, 2014.

[7] N. Daoust, Requirements Modeling for Bussiness Analysts. Technics Publications, LLC, 2012.

[8] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in International Conference on Software Engineering, FOSE, 2007, pp. 37–54.

[9] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, “A framework for testing uml activities based on fuml,” in Proc. of 10th Int. Workshop on Model Driven Engineering, Verification, and Validation, vol. 1069, 2013, pp. 11–20.

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

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