Simulation Modelling Practice and Theory 74 (2017) 1–16

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Generating event logs for high-level process models[☆]

Alexey A. Mitsyuk[a], Ivan S. Shugurov[a], Anna A. Kalenkova[a,*], Wil M.P. van der Aalst[b,a]

[a] National Research University Higher School of Economics, Laboratory of Process-Aware Information Systems, Moscow, 101000, Russia
[b] Eindhoven University of Technology, Department of Mathematics and Computer Science, PO Box 513, NL-5600 MB Eindhoven, The Netherlands

ARTICLE INFO

Article history:
Received 20 May 2016
Revised 22 January 2017
Accepted 24 January 2017

Keywords:
Event log generation
Business process simulation
BPMN (Business process model and notation)
Process mining

ABSTRACT

Business Process Model and Notation (BPMN) is a de-facto standard for practitioners working in the Business Process Management (BPM) field. The BPMN standard [1] offers high-level modeling constructs, such as subprocesses, events, data and message flows, lanes, and is widely used to model processes in various domains. Recently several BPMN-based process mining techniques [2, 3, 4] were introduced. These techniques allow representing processes, discovered from the event logs of process-aware information systems, in a convenient way, using the BPMN standard. To test these mining approaches an appropriate tool for the generation of event logs from BPMN models is needed. In this work we suggest such a tool. We propose a formal token-based executable BPMN semantics, which takes into account BPMN 2.0 with its expressive constructs. The developed tool is based on these semantics and allows simulation of hierarchical process models (including models with cancellations), models with data flows and pools, and models interacting through message flows. To manage the control flow, script-based gateways and choice preferences are implemented as well. The proposed simulation technique was implemented on top of existing plug-ins for ProM (Process Mining Framework) [5], and was verified on models created by practitioners from various domains.

1. Introduction

BPMN (Business Process Model and Notation) 2.0 [1] is one of the most frequently used process modeling notations and is a de facto standard in business process modeling. Petri nets are respected in academia due to their simplicity, ability to express concurrency, clear semantics, and mathematical nature. These notations have good instrumental support and are widely used by both researchers and practitioners.

Process mining [6] is a research area which offers tools and methods for analysing and improving processes by looking at insights hidden in event logs of information systems. In addition to Petri nets, which remain the most frequently used modeling notation among process miners, process mining results are increasingly visualized as BPMN models [2–4].

Most of process mining algorithms require an event log as an input parameter. However, in order to develop and systematically evaluate these algorithms one requires large sets of models and corresponding event logs. Although the goal of

☆ This work is supported by the Basic Research Program at the National Research University Higher School of Economics.
* Corresponding author.
E-mail addresses: amitsyuk@hse.ru (A.A. Mitsyuk), shugurov94@gmail.com (I.S. Shugurov), akalenkova@hse.ru (A.A. Kalenkova), w.m.p.v.d.aalst@tue.nl (W.M.P. van der Aalst).

http://dx.doi.org/10.1016/j.simpat.2017.01.003
1569-190X/© 2017 Elsevier B.V. All rights reserved.

process mining is to analyze real-life event logs, they are often not suitable for the verification of new algorithms at early stages of the development. The potential problem is to find (or construct) a log with particular characteristics. Artificially generated logs are needed for better testing and evaluation of the process mining algorithms. The possible evaluation approach can be defined as follows. First, one generates an event log by simulating the selected process model. Second, he or she applies a novel process discovery algorithm to this log. Finally, the discovered model is compared to the initial one. Note that in real-life event logs such a "ground truth" is often missing. Therefore, it is important to be able to generate models and logs in a controlled manner.

Our objective is to design and implement a concrete approach for the generation of event logs by direct simulation of BPMN models. We start by discussing related work (Section 2), and introducing a running example (Section 3). The three *main contributions* are described in Sections 4–6 respectively. In particular, Section 4 introduces a formal BPMN 2.0 semantics, Section 5 presents event log generation algorithms. The reader can find a description of the tool implementing the proposed algorithms and its evaluation in Section 6.

2. Related work

Several papers have been published in the field of artificial event log generation in the context of process mining. In this section we will take a look at existing approaches and tools.

Manual generation of logs with particular characteristics is the most naive way to test process mining algorithms. However, in some cases it is useful. For example, during implementation of a discovery algorithm one usually has several very simple samples to evaluate the code in very straightforward situations. Manual generation has evident limitations and disadvantages. Creating several larger sets of logs through manual generation is extremely tedious and possibly leads to many of mistakes. Usually, it is also a very time-consuming activity.

An approach for the generation of artificial event logs using *CPN Tools* has been proposed in [7]. *CPN Tools* is a widely used colored Petri nets editor with powerful simulation and analysis abilities. The proposed extension for *CPN Tools* provides an opportunity to generate random logs based on a given Petri net. The main difficulty of the approach is that it implies writing scripts in the Standard ML language, which leads to possible problems during tool adaptation for a specific task. *CPN Tools* does not support the simulation of BPMN models directly. Although manual approaches for transformation of BPMN subsets to CPN were discussed in [8,9], there are no well-defined and implemented algorithms to perform these transformations. Thus, a lot of manual work is needed to simulate BPMN models, which includes control-, data-flow, messages, and other BPMN-specific concepts using the colored Petri nets notation.

Yet another way to simulate BPMN models is to follow a two-step approach: (1) transform a BPMN model to a modeling formalism called DEVS (*Discrete Event System Specification* [10]), (2) then simulate DEVS model using one of the DEVS simulation tools [11]. Different implementations of such an approach are considered in [12,13]. In these papers authors present two software tools for BPMN-to-DEVS transformation. Another BPMN-to-DEVS transformation is presented in [14].

In comparison to both these approaches, our method works without an additional transformation step. We state that BPMN models can be executed directly. Moreover, we present a corresponding semantics for BPMN models and a plug-in for ProM Framework [5], which implements the proposed approach. Since ProM provides integration of various process discovery and log processing plug-ins, the direct simulation based on the formal BPMN semantics can be incorporated with other process mining approaches to automate testing of BPMN discovery methods.

Yet another instrument for event logs generation is *SecSy tool* [15]. The purpose of this tool is to generate artificial event logs for testing algorithms in the field of security-oriented information systems modeling. The tool generates an event log satisfying the behavior predefined by a given specifications. However, SecSy mainly focuses on security-oriented aspects of information systems.

Processes Logs Generator (PLG, and its later version PLG2) is a highly configurable toolbox to produce event data. PLG [16,17] is a framework for the generation of artificial business process models in a form of dependency graphs and event logs of their execution. The authors employ context-free grammars to build process models. A similar technique for generating structured BPMN models and event logs is used in [18]. These approaches deal only with block-structured process models.

Event logs generation is also possible using the well-known *BPMN engines* (Activiti[1], Bizagi[2], and others). These frameworks are developed to maintain real-life business processes using Java or other technologies, and can be extended with log generation functions. Unfortunately, these engines usually have rather complex architecture. Thus, it is hard to apply them in research. Moreover, engines usually do not follow the BPMN standard strictly. For example, the synchronization of parallel branches in Activiti (Camunda) differs from the specification. The current version of the Bizagi engine can not simulate sub-processes with cancellations and is not an open source tool, thus, it can not be modified. iGrafx[3] is a set of solutions for process management, analysis, and collaborative process design. Among others it has a simulation functionality. The notation used in it is similar to BPMN, but formal translation rules are not presented. The simulator fits the goals of statistical analysis, but has no functions for flexible event log generation.

[1] Activiti BPM platform: http://activiti.org/.
[2] Bizagi engine: http://www.bizagi.com/en/.
[3] iGrafx platform: http://www.igrafx.com.

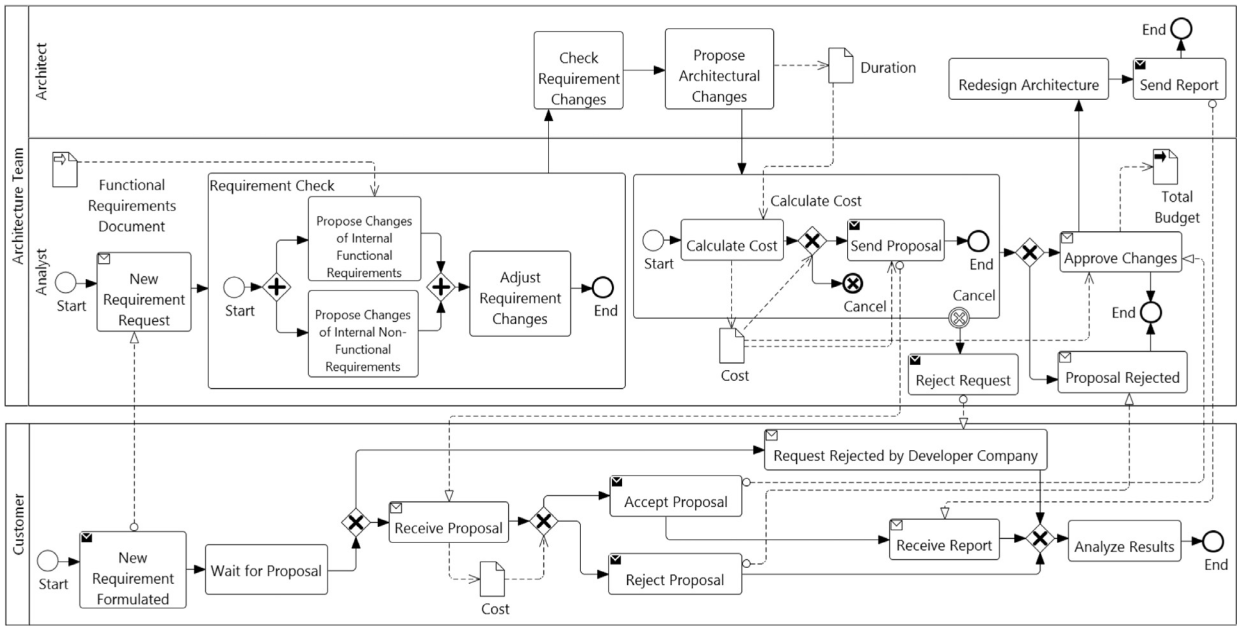


Fig. 1. Architectural changes process.

This paper focuses on the generation of event logs through simulation. There are also approaches that aim to transform event data into simulation models. See [19] for example. Another related paper on mixing simulation with process mining is [20]. Characteristics of generated event logs were investigated in [21,22]. Authors of [23] investigate problems of dealing with large model collections.

The approach (and software) presented in this paper is an extension of the earlier one proposed in [24]. The previous version deals only with process models in the classical Petri net notation. The new version of the approach presented in this paper uses a well-defined subset of the BPMN standard as a language for process models. The user can generate logs in a flexible manner using any BPMN model containing the main modeling elements (activities, gateways, cancellations, message flows, and nested processes). One can fine-tune the experiment by using a predefined process model. Another significant improvement is a new portion of data-flow-oriented features. The user can specify behavior of each particular node connected with data objects by assigning a script written in Python.

In contrast to the log generators proposed earlier, our tool supports: (1) generation of event logs using (possibly non-block-structured) BPMN models made by experts in any modeling tool; (2) cancellation events and message flows; (3) data-driven gateways with assigned scripts; (4) BPMN pools and lanes considered as process roles; (5) time.

3. Motivating example

Let us consider an example of a high-level process model, which can be simulated with the approach proposed in this paper. Modern best practices in software and system engineering assume application of well-defined development and design processes. Fig. 1 shows a model of a requirement change process.

The model contains two main *pools* related to the actors (roles) involved in the process: *Customer* and *Architecture Team*. During an iteration of the development process the Customer formulates *new requirements* and sends it to a company's *Architecture Team* via a message flow. We consider only architects and analysts as architecture team members (they are represented as *lanes*). Firstly, the customer's request is checked by an analyst in a nested process called *Requirement Check*. A formal list of changes prepared is moved to an architect. The architect *proposes architectural changes* to satisfy new requirements and estimates time needed to implement them, which is stored in the *Duration* data object. The analyst *calculates the costs* of an implementation, and sends a special proposal for the customer via message flow. If calculated *costs* are very high or low to meet the business rules of the developer company, the project is rejected. Such a case can be modeled using *cancellation end* and *intermediate boundary events*. The customer can *accept* or *reject* the proposal. If the customer accepts the proposal, the analyst *approves changes*, and updates the total budget of the project via outgoing data object called *Total Budget*. The architect *redesigns architecture* and *sends report* to the customer via message flow. Then the customer *analyzes the results*. The same analysis is performed in both reject cases.

Such high-level models are quite common for the BPM field. They have relatively complex structures and use various BPMN elements. Automated analysis and discovery of such models can give useful insights. While such discovery algorithms are being actively developed, researchers face the problem of testing them. However, as it was discussed in Section 2, there are no tools supporting all modeling elements shown in Fig. 1 in a flexible manner. Developing such tools is the main motivation for the work presented in this paper.

4. Event logs and BPMN modeling constructs

This section presents event logs, BPMN modeling constructs and their semantics underlying the proposed log generation algorithms. BPMN offers a wide range of modeling elements but not all of them are frequently employed [25]. In contrast to other proposed semantics [26–28], in this paper we consider key BPMN constructs which cover the main workflow perspectives: control, resource, and data. Furthermore, the time perspective is considered as well. The set of selected modeling constructs includes elements which can be mined using existing process discovery algorithms [2–4], and those of high interest for process miners [6,29].

First, we introduce multisets used to define states of BPMN models and event logs, in which one trace can appear multiple times. *Multiset* $b : A \to \mathbb{N}$ over set A contains element $a \in A$, $b(a)$ times. A set of all multisets over A is denoted as $\mathcal{B}(A)$. We will consider sets as a special case of multisets, where elements can appear 0 or 1 times.

Function f: $X \nrightarrow Y$ is called a *partial function* with domain $dom(f) \subseteq X$ and range defined as $rng(f) = \{f(x) | x \in dom(f)\} \subseteq Y$. If $dom(f) = X$, then *f is a total function*.

4.1. Event logs

Event logs are used in almost all process mining approaches. Event logs show the behavior (at least a part of it) of an information system being investigated. This is the only way to touch real life for analysts.

An event contains a name and can be equipped with additional attributes such as: resource, timestamp, and different data variables. Let us define some basic notions which will be used in later definitions. Let $\mathcal{U}_A$ be a set of possible activity labels (event names), $\mathcal{U}_{Attr}$, and $\mathcal{U}_{Val}$ be sets of possible attributes and values correspondingly [30].

Definition 1 (Event log)**.** $L = (E, T, act, attr)$ is an event log where:

- E is a set of events,
- T is a set of traces where each trace is a sequence of events, i.e., $T \subseteq E^*$,
- $act : E \to \mathcal{U}_A$ is a function which maps each event onto its corresponding activity (event name),
- $attr : E \to (\mathcal{U}_{Attr} \nrightarrow \mathcal{U}_{Val})$ is a function which defines event attributes and their values.

Hence, an event log is a set of traces. A trace $\langle e_1, e_2, \dots, e_n\rangle \in T$ is a sequence of events. Each event e corresponds to activity $act(e) \in \mathcal{U}_A$ and has a set of attributes *dom(attr(e))*. One of the standard attributes is *time*, i.e., *time* $\in$ *dom(attr(e))* and *attr(e)(time)* is the *timestamp* of *e*. Another standard attribute is *resource: resource* $\in$ *dom(attr(e))* and *attr(e)(resource)* is a name of the performer of *e*. Attributes are also used to record the values of data objects, such attributes have names of corresponding data objects. Note that in this paper we assume each event appear only once in the event log.

4.2. Base BPMN modeling constructs

A typical model of a process control flow contains activities, start/end events, routing elements, and connections between them. We also add a resource perspective which specifies performers of the activities. Usually resources are represented as BPMN lanes.

According to the BPMN 2.0 standard activities, which represent elementary steps of the process, are called tasks. So-called gateways and sequence flows are used to connect activities and to express different types of control-flow routing concepts. Formally, we define a base BPMN model as follows.

Definition 2 (Base BPMN model)**.** A base BPMN model is a tuple: $(N, A, G_{XOR}, G_{AND}, e_{start}, E_{end}, E_{cancel}, SF, EF, \lambda_a, R, res)$, where

- N is a set of flow nodes,
- $A, G_{XOR}, G_{AND}, \{e_{start}\}, E_{end}$ form a partition of N,
- $A \subseteq N$ is a set of activities,
- $G_{XOR} \subseteq N, G_{AND} \subseteq N$ are sets of exclusive and parallel gateways respectively,
- $e_{start} \in N$, $E_{end} \subseteq N$, $E_{cancel} \subseteq E_{end}$, such that $E_{end} \backslash E_{cancel} \neq \emptyset$ are a start event, a set of end events, and a set of cancellation end events respectively,
- $SF \subseteq N \times N$ is a set of sequence flows, $EF \subseteq SF \cap (A \times N)$ is a set of exceptional flows, such that $\forall a \in A\ \exists n \in N$: $(a, n) \in SF$, $(a, n) \notin EF$, i.e., each activity has a non-exceptional outgoing sequence flow,
- $\lambda_a : A \to \mathcal{U}_A$ is a labeling function,
- R is a set of resources, $res : A \nrightarrow R$ is a partial function which maps some of the activities onto set of resources.

This definition is based on the definition of Labeled BPMN Model from [3]. Note, that the BPMN 2.0 standard [1] describes more gateway and event types. In particular, there are the following gateway and intermediate event types: complex, inclusive, event-based, parallel event-based gateways; message, timer, link, signal, multi, error, escalation and other events. We do not consider these elements in our base BPMN model. There are two reasons for that. First, our token-based semantics

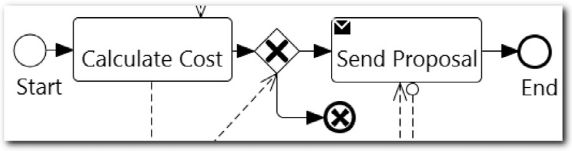


Fig. 2. A fragment with cancellation.

is supported by gateways with local dependencies, i.e., the enablement of a gateway depends only on its ingoing flows. This is the case for the gateway types we consider. Second, many of the interrupting events behaviorally correspond to the cancellation event descried in this work. In the future we plan to extend the set of supported BPMN elements.

An example of a base BPMN model is shown in Fig. 2. This is a fragment of the example process shown in Fig. 1. The fragment describes a cost calculation for a particular proposal. Depending on the cost, a proposal can be either sent to the customer, or canceled.

The *start event* e_{start} is an event without any incoming sequence flows. *End events* E_{end} are events without any outgoing sequence flows. All events are shown as circles. End events are represented by circles with bold borders. Cancellation events are marked with an additional “x” sign.

A *marking* of a BPMN model denotes a current state of a process instance. It is a multiset over the set of sequence flows, denoted as $m : SF \to \mathbb{N}$. That is, each sequence flow may contain tokens or not. A token in some sequence flow shows, that the source activity (or gateway) of this sequence flow is already fired, whereas the target activity (gateway) may fire, if all other incoming flows for it also carry tokens. A marking shows a snapshot of the process instance execution at a particular moment in time. An *initial marking* is a marking such that $m(sf) = 1$, if sf is an outgoing sequence flow of e_{start}, $m(sf) = 0$ otherwise. *Final marking* is a marking such that $\forall sf \in SF : m(sf) = 0$. Each node may be enabled, then an enabled node may fire.

An *activity* is depicted as a rounded rectangle and denotes some visible action, which occurs in a business process. The fragment shown in Fig. 2 contains two activities “Calculate Cost” and “Send Proposal”. The first activity is performed in all process instances, whereas the branch with the latter one can be cancelled. An activity $a \in A$ is said to be in enabled state in marking m if at least one of its incoming sequence flows holds at least one token. An enabled activity a may fire. When activity a fires it consumes one token from an incoming sequence flow and produces a token for each outgoing not exceptional sequence flow.

Gateways of two types (parallel and exclusive) are needed to redirect control flow during process runs. *Exclusive gateways* (shown as diamonds with an “x” sign) merge alternative paths: incoming sequence flow token is routed to one of the outgoing sequence flows. A *parallel gateway* (modeled as a diamond with a plus sign) is enabled if each incoming sequence flow contains at least one token. An enabled parallel gateway may fire, consuming a token from each incoming sequence flow and producing a token to each outgoing sequence flow. The reader can see an exclusive gateway in Fig. 2.

An *end event* is enabled if one of its incoming sequence flows contains a token. When an ordinary end event fires, it consumes a token from an incoming sequence flow, while a *cancellation end event* consumes all the existing tokens, yielding an empty marking.

Process resources are modeled with *lanes*. The reader can see in Fig. 1, that the process of the architecture team is performed by two following roles: “Analyst” and “Architect”. The lanes are separated with a solid line and placed in a single BPMN *pool*.

4.3. Hierarchical BPMN models

BPMN supports hierarchy, i.e., a process can contain sub-processes. Sub-processes can be presented as base BPMN models.

Fig. 3 shows a fragment of the architectural changes process model (see Fig. 1) with sub-process “Calculate Cost”. Sub-process is an expanded activity of the base model, which contains a nested model. First, the atomic activity “Calculate Cost” is performed. After its completion either the prepared proposal is sent to the Customer, or the proposal is canceled. Note that a sub-process can be either finished successfully by reaching an end event, or canceled. The choice is made by the exclusive gateway. This situation is modeled using a cancellation event. The outgoing exceptional flow is marked with a boundary intermediate cancellation event.

Let us introduce a formal definition for hierarchical models.

Definition 3 (Hierarchical BPMN model)**.** A hierarchical BPMN model is a tuple: $BPMN_h = (BPMN_{models}, BPMN_{root}, H, ref, cancel)$, where

- $BPMN_{models} = \{BPMN_1, \ldots, BPMN_m\}$ is a set of base BPMN models[4],

[4] Let $A = A_1 \cup \ldots \cup A_m$, $EF = EF_1 \cup \ldots \cup EF_m$, and $E_{cancel} = E_{cancel_1} \cup \ldots \cup E_{cancel_m}$, where A_1, ... , A_m, EF_1,... , EF_m, and E_{cancel_1},... , E_{cancel_m} are sets of activities, exceptional flows, and cancellation events of base models $BPMN_1$,... ,$BPMN_m$ respectively.

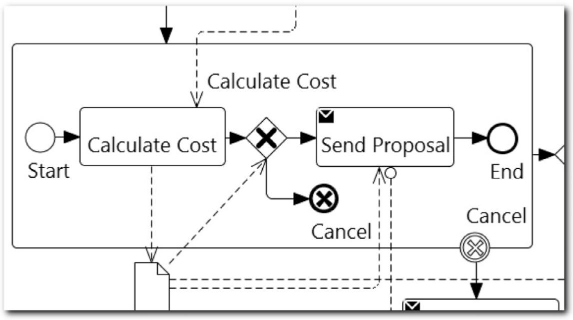


Fig. 3. A fragment with sub-process.

- $H \subseteq BPMN_{models} \backslash \{BPMN_{root}\} \times BPMN_{models}$ is a tree relation, which specifies a parent for each BPMN model, where $BPMN_{root}$ is a root of this tree,
- $ref : A \nrightarrow BPMN_{models} \backslash \{BPMN_{root}\}$, such that $(BPMN_{child}, BPMN_{parent}) \in H$ iff $\exists a \in A_{parent} : ref(a) = BPMN_{child}$, where A_{parent} is a set of $BPMN_{parent}$ activities, i.e. *ref* specifies the link between activities in a parent model and child models, there is an activity for each child model,
- $cancel : E_{cancel} \rightarrow EF$ is a bijective function, such that it maps cancellation end events of $BPMN_{child}$ to exceptional sequence flows of $BPMN_{parent}$ for each pair $(BPMN_{child}, BPMN_{parent}) \in H$.

Consider a hierarchical BPMN model $BPMN_h = (BPMN_{models}, BPMN_{root}, H, ref, cancel)$. We say that activities, for which *ref* function is not defined, are *atomic activities*. All other activities are *non-atomic*. A set of *nested process models* for a model $BPMN_{parent} \in BPMN_{models}$ is defined as $\{BPMN_{nested} \in BPMN_{models} \ |(BPMN_{nested}, BPMN_{parent}) \in H^*\}$, where H^* is a transitive closure of *H*.

Suppose that e_{start} is a start event of model $BPMN_{root}$. Then an *initial marking m* of $BPMN_h$ is defined as follows: $\forall sf \in SF :$ $m(sf) = 1,$ if *sf* is an outgoing sequence flow of e_{start}, and $m(sf) = 0$, otherwise.

Firing rules of hierarchical BPMN models extend the firing rules of base BPMN models in a part of enabling and firing non-atomic activities. Suppose that *a* is a non-atomic activity of a base model $BPMN_{parent} \in BPMN_{models}$, then exists $BPMN_{child} \in BPMN_{models}$, such that $ref(a) = BPMN_{child}$. Activity *a* can fire if and only if $BPMN_{child}$ and its nested models do not contain any tokens. When activity *a* fires it consumes a token from an incoming sequence flow and produces tokens to outgoing sequence flows of $BPMN_{child}$ start event.

If $BPMN_{child}$ reaches the final marking and a cancellation end event *n* of $BPMN_{child}$ is the cause of termination, then a token will be added to *ef* – an exceptional sequence flow of $BPMN_{parent}$, such that $cancel(n) = ef$, and all the tokens will be removed from sequence flows of nested models. Otherwise, if $BPMN_{child}$ and its nested models reach the final markings, then tokens will be added to outgoing not exceptional sequence flows of *a*.

The BPMN notation contains a wide range of event constructs, the semantics of which involves cancellation. These could be error, signal, cancel, and other types of events. In this paper we combine all of them together conceptually as one type called *cancel* events.

4.4. BPMN models with data

In this subsection we will enrich BPMN models with data objects and data associations. Fig. 4 shows a fragment of architectural changes process model (see Fig. 1) with the data object “Cost”. Data associations are shown as dashed arrows. The model contains an explicit choice between acceptance and rejection of an architectural changes proposal, prepared by the architecture team as a result of analysis of customer’s needs. Then it was received by the customer (“Receive Proposal”

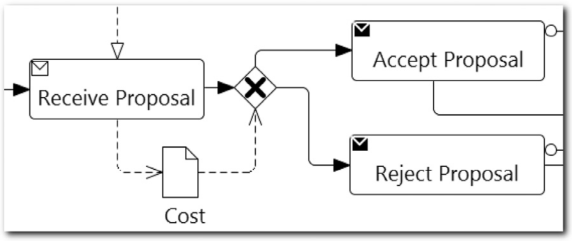


Fig. 4. A fragment with data object.

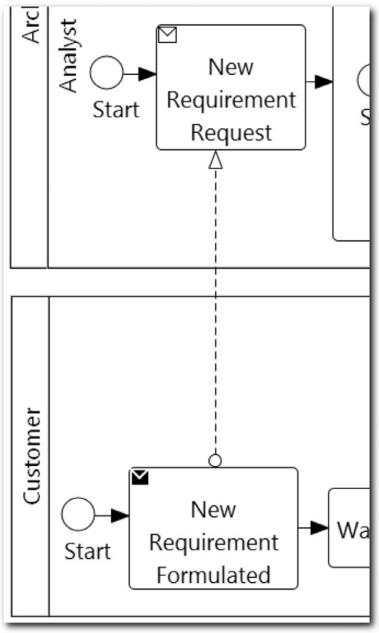


Fig. 5. A fragment with participants.

activity in Fig. 4). The choice is made in an exclusive gateway, basing on the results of the analysis performed earlier. These results stored in *data object* “Cost” shown with a symbolic sheet of paper. It has two following *data associations: write* association with activity “Receive Proposal” and *read* association with the exclusive gateway. Thus, the cost of received proposal is defined as a value of the data object after the firing of “Receive Proposal” activity. If cost is not appropriate, the gateway activates the “Reject Proposal” branch. In other case, the process proceeds with the “Accept Proposal” activity.

Definition 4 (BPMN model with data)**.** A BPMN model with data is a tuple $BPMN_{data} = (BPMN_h, DO, DA, f_A, f_{SF}, default, \lambda_d)$, where:

- $BPMN_h$ is a hierarchical BPMN model, where $BPMN_{models} = \{BPMN_1, \ldots, BPMN_m\}$ is a set of base BPMN models[5],
- $DO = \{do_1, \ldots, do_k\}$ is a set of data objects,
- $DA \subseteq (A \times DO) \cup (DO \times A) \cup (DO \times G_{XOR})$ is a set of data associations,
- $f_A : A \to ((DO \to \mathcal{U}_{Val}) \to (DO \to \mathcal{U}_{Val}))$ is a function which specifies how activities change the values of data objects,
- $f_{SF} : SF' \to ((DO \to \mathcal{U}_{Val}) \to \{true, false\})$ where $SF' = \{(g, n) \in SF' | g \in G_{XOR}, n \in N\}$ is a function, which defines conditional branching,
- $default : G_{XOR} \to SF$ is a function which specifies a default outgoing sequence flow $sf \in SF$ for each exclusive gateway $g_{XOR} \in G_{XOR}$,
- $\lambda_d : DO \to \mathcal{U}_{Attr}$ is a function, which maps each data object to its name.

A *marking* of a BPMN model with data is defined as a pair (m, d), where $m : SF \to \mathbb{N}$ is a marking of hierarchical model $BPMN_h$, and $d : DO \to \mathcal{U}_{Val}$ presents values of data objects.

BPMN models with data extend firing rules of hierarchical BPMN models. When an activity $a \in A$ fires in a marking (m, d) it defines and sets new data object values: $d' = f_A(a)(d)$. Firing of an exclusive gateway g_{XOR} in a marking (m, d) results in producing a token to one of the outgoing sequence flows $sf \in SF$, for which $f_{SF}(sf)(d) = true$. If for all outgoing sequence flows sf holds that $f_{SF}(sf)(d) = false$, a token is added to the flow $default(g_{XOR})$.

4.5. Interacting BPMN models

In this subsection we will introduce BPMN models, which represent different participants, interacting through message flows. In Fig. 5 the reader can see a fragment of the architectural changes process model (see Fig. 1), in which new requirements are sent as messages from a customer to the architecture team. The message flow is shown as a dashed line with a white arrow at the side of a message receiver (activity “New Requirement Request”) and a white circle at the side of a sender (activity “New Requirement Formulated”).

[5] Further we will use the following notations: $N = N_1 \cup \ldots \cup N_m$, $A = A_1 \cup \ldots \cup A_m$, $G_{XOR} = G_{XOR_1} \cup \ldots \cup G_{XOR_m}$, $SF = SF_1 \cup \ldots \cup SF_m$, where $N_1, \ldots, N_m, A_1, \ldots, A_m, G_{XOR_1}, \ldots, G_{XOR_m}, SF_1, \ldots, SF_m$ are sets of nodes, activities, exclusive gateways and sequence flows of models $BPMN_1, \ldots, BPMN_m$ respectively.

Each participant is usually visualized as a pool within the entire BPMN model[6].

Definition 5 (BPMN model with participants)**.** A BPMN model with participants is a tuple $BPMN_{part} = (BPMN_p, MF)$, where

- $BPMN_p = \left\{BPMN_{d_1}, \dots, BPMN_{d_k}\right\}$ is a set of BPMN models with data,
- $MF \subset (A \times A)\backslash(A_1 \times A_1)\backslash \dots \backslash(A_k \times A_k)$ [7] is a set of message flows.

A marking of a BPMN model with participants $BPMN_{part} = (BPMN_p, MF)$ can be represented as $(m_1, \dots, m_k, m')$, where m_i is a marking of $BPMN_{d_i}$ model for $i \in \overline{1, k}$, and $m' \in (MF \to \mathbb{N})$. An initial marking is represented as $(m_{1_0}, \dots, m_{k_0}, m_0')$, where m_{i_0} is an initial marking of a corresponding $BPMN_{d_i}$ model.

Message flows extend the firing rules of activities. In addition to the standard enabling conditions presented earlier, each incoming message flow of an activity must contain at least one token to make this activity enabled, i.e., activity "New Requirement Request" in Fig. 5 will become enabled only when some request will be sent from the customer. When an activity fires, apart from the standard operations with tokens on sequence flows, it consumes a token from each incoming message flow and produces a token to each outgoing message flow.

BPMN pools and messages allow to model and simulate systems with multiple participants working in parallel and can be synchronized via message flows. This is the only modeling element allowed to cross pool's boundary. Note, that in Fig. 1 the processes of customer and architecture team have independent starts and ends. Message flows is the only mechanism that binds them together.

5. Event log generation

In this section we present our approach for the generation of event logs. We will consider the different types of BPMN models described earlier. The remainder of the section is organized as follows: first, an algorithm for generating event logs from base BPMN models is described; then each next section explains a specific enhancement of the algorithm.

5.1. A basic approach for event log generation

This subsection is devoted to an algorithm used for the simulation of base BPMN models as they are defined above. An event log corresponds to Definition 1. Recall that we assume each event appear one in the log. Each trace corresponds to an execution of a single process instance. That is, all tokens in a model created during generation of one trace relate to the same instance of a process being simulated.

Algorithm 1 presents a base simulation approach. During a model execution it constructs a set of event logs, each of which consists of traces.

Firing of an activity leads to the creation of an event with certain attributes, such as its name and resource. If lanes are presented in a model, they are used during the generation in order to specify resources involved in the execution of activities.

In Algorithm 1 all events and gateways are fired first, and only then activities are executed. Before checking whether activities are enabled, all possible firings of gateways are carried out[8]. After that we have the entire set of activities enabled at a particular step of simulation. Thus, each of them can fire with the equal probability. Due to the invisible nature of gateways execution, it is enough to find the first enabled gateway and then it can be immediately fired. By invisible nature we mean, that there are no records in the event log corresponding to the gateway firings. They are needed for the control-flow management only. In contrast, activities require more careful handling. First, all enabled activities are found and only then a random one is fired. Outgoing sequence flows of fired exclusive gateways are selected randomly as well. Therefore, there are two stages in simulation. First, all events and gateway are fired in an arbitrary order until the only enabled elements are activities. Second, one of the enabled activities is selected randomly and fired. This ensures equal probability of firing of each enabled activity at each step. Moreover, the choice of one or another process model branch is also made randomly (Algorithm 2).

The generation of a trace terminates when one of two situations occur: (1) there are no more tokens in the model; (2) execution exceeded the specified maximal number of firings. In the former case the replay finished successfully (process instance is completed), the generated trace is added to the event log. In the latter case, the replay did not terminate properly and the trace will not be added to the event log.

Forced termination is supported to deal with very large or live-locking models.

[6] Note that in contrast to lanes sequence flows cannot cross pools borders.

[7] $A = A_1 \cup \dots \cup A_k$ and A_1, ... ,A_k are disjoint sets of all activities of the base BPMN models, forming models $BPMN_{d_1}$,... ,$BPMN_{d_k}$ respectively.

[8] Note, that there can be a cyclic firing of gateways. In that case according to Algorithm 1 the log generation process will be terminated and the trace will not be added to a log.

```
Input: $BPMN_{model} = (N, A, G_{XOR}, G_{AND}, e_{start}, E_{end}, E_{cancel}, SF, EF, \lambda_a, R, res)$, $N_l$ is a number of logs to generate, $N_t$ is a maximal number of traces in each log, and $N_f$ is a maximal number of firings per trace
Output: a set of event logs $L = \{L_1, \ldots, L_{N_l}\}$, where $L_i = (E_i, T_i, act_i, attr_i)$, $i \in \overline{1..N_l}$
function GENERATE($BPMN_{model}$, $N_l$, $N_t$, $N_f$)
    $L \leftarrow \{\}$;
    for $i \leftarrow 1$ up to $N_l$ do
        $E_i \leftarrow \{\}$;
        $T_i \leftarrow \{\}$;
        $act_i \leftarrow \{\}$;
        $attr_i \leftarrow \{\}$;
        $L_i \leftarrow (E_i, T_i, act_i, attr_i)$;
        for $j \leftarrow 1$ up to $N_t$ do
            $trace \leftarrow \langle\rangle$;
            $firing \leftarrow 0$;
            // $m_0$ - is an initial marking of $BPMN_{model}$
            $m \leftarrow m_0$;
            repeat
                $firedNode \leftarrow false$;
                for $\forall n \in G_{XOR} \cup G_{AND} \cup E_{END}$ do
                    if $n$ is enabled then
                        $m \leftarrow fire(n, m, BPMN_{model})$;
                        $firedNode \leftarrow true$;
                        break;
                    end if
                end for
                if not $firedNode$ then
                    $activities \leftarrow \{a \in A | a \text{ is enabled}\}$;
                    if $activities = \emptyset$ then
                        // deadlock is reached, terminate execution
                        $firing \leftarrow N_f$;
                    else
                        // select a n activity to fire
                        $a \leftarrow choose(activities)$;
                        EXECUTE($a$, $BPMN_{model}$, $m$, $L_i$);
                    end if
                end if
                $firing \leftarrow firing + 1$;
            until $firing < S_f$ and $\exists sf \mid sf \in SF$ and $m(sf) > 0$
            if $\forall sf \in SF$ $m(sf) = 0$ then
                $T_i \leftarrow T_i \cup \{trace\}$;
            end if
        end for
        $L \leftarrow L \cup L_i$;
    end for
    return $L$;
end function
```

Algorithm 1. A basic event logs generation.

```
procedure EXECUTE($a$, $BPMN_{model}$, $m$, $L_i = (E_i, T_i, act_i, attr_i)$)
    $m \leftarrow fire(a, m, BPMN_{model})$;
    $e \leftarrow createEvent()$;
    $E_i \leftarrow E_i \cup \{e\}$;
    $T_i \leftarrow T_i + e$;
    $act_i \leftarrow act_i \cup \{(e, \lambda_a(a))\}$;
    if res (a) is defined for $a$ then
        $attr_i \leftarrow attr_i \cup \{(e, (resource, res(a))\}$;
    end if
end procedure
```

Algorithm 2. Algorithm 1 continued.

5.2. Generating event logs for BPMN models with data

In contrast to the simulation of hierarchical and interacting BPMN models, which have additional firing rules and thus extend the basic log generation algorithm in a natural way, simulation of a BPMN model with data should be described separately.

Algorithm 3 extends the basic Algorithm 1 with a modified procedure of activity execution: each firing of an activity not only changes the marking, but assigns novel values to data objects. The values of data objects are written to the log in a form of event attributes.

Algorithm 3 Event generation with data.

```
Input: activity $a$, $BPMN_{data} = (BPMN_h, DO, DA, f_A, f_{SF}, default, \lambda_d)$, $(m, ((do_1, val_1), \ldots, (do_n, val_n)))$, $L_i = (E_i, T_i, act_i, attr_i)$
1: procedure EXECUTE($a$, $BPMN_{data}$, $L_i$)
2:     // $\lambda_a$, $res$, $ref$ are functions defined by
3:     // BPMN models, which contain activity $a$
4:     $m \leftarrow fire(a, m, BPMN_{data})$;
5:     if $ref(a)$ is not defined then
6:         $e \leftarrow createEvent()$;
7:         $E_i \leftarrow E_i \cup \{e\}$;
8:         $T_i \leftarrow T_i + e$;
9:         $act_i \leftarrow act_i \cup \{(e, \lambda_a(a))\}$;
10:        if res (a) is defined for $a$ then
11:            $attr_i \leftarrow attr_i \cup \{(e, (resource, res(a))\}$;
12:        end if
13:        for $do_j \in DO$ do
14:            $attr_i \leftarrow attr_i \cup \{(e, (\lambda_d(do), f_A(a)((do_1, val_1), \ldots, (do_n, val_n))(do_j))\}$;
15:        end for
16:    end if
17: end procedure
```

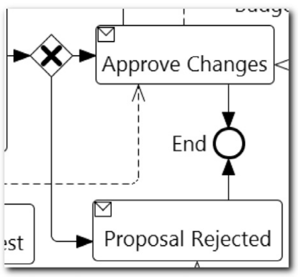


Fig. 6. A model fragment with an exclusive gateway.

Additionally, as it follows from the definition of BPMN models with data, during the execution of an exclusive gateway, a token is added to one of the outgoing sequence flows, for which the value of a special assigned function is *true*. If the values of functions for all outgoing sequence flows are *false*, a token is added to the *default* outgoing sequence flow.

5.3. Simulating BPMN models with preferences

In order to make simulation more realistic we introduce a notion of so-called *preferences*, which is used to modify the choice between conflicting activities. For the BPMN language that is the case when dealing with exclusive gateways.

A typical scenario is shown in Fig. 6, which is also a fragment of Fig. 1. Obviously, in a real-life setting, rejections and approvals happen with different frequencies. Thus, sometimes it seems rational to set one activity as more frequent than the other when generating an event log. For example, one may want to simulate the situation, when almost all proposals are approved.

A preference is a non-negative integer value within some range. Each preference corresponds to exactly one outgoing sequence flow. The higher a preference is, the higher possibility to add a token to a certain sequence flow is. The precise formula for the *probability of a sequence flow to get a token* is $P(s_j) = \frac{pref(s_j)}{\sum_{i=1}^{|OSF|} pref(s_i)}$. Where *OSF* is a set of outgoing sequence flows from a specific exclusive gateway; $s_i, s_j \in OSF$ are two outgoing sequence flows, and *pref(s)*, $s \in OSF$ is a preference of the particular sequence flow s from the set *OSF*.

For the case shown in Fig. 6 *OSF* contains two sequence flow to activities “Approve Changes” and “Proposal Rejected”. For example, one can set $pref(s_{ApproveChanges}) = 10$ and $pref(s_{ProposalRejected}) = 1$. Then, changes will be approved in most cases.

Note that it is possible to completely disallow one or more outgoing flows in order to achieve the desired behavior. However, it is ensured that at least one outgoing sequence flow has a non-zero preference: $\sum_{i=1}^{|OSF|} pref(s_i) > 0$.

The algorithm used for checking whether an exclusive choice gateway is enabled remains intact – we only need to extend the way an outgoing sequence flow is selected. The pseudo code is provided in Algorithm 4.

5.4. An approach for generating event logs with time

In order to make simulation more similar to the way processes are executed in real life, where activities take time to be performed, a slight extension of the notion of token is required.

Algorithm 4 An outgoing sequence flow selection.

```
Input:an exclusive choice gateway $g \in G_{XOR}$ Output:an outgoing sequence flow of $g$
1: function SELECT($g$)
2:     //g• is a list of outgoing sequence flows for g,
3:     if size(g•) = 1 then
4:         return $g \bullet [1]$;
5:     end if
6:     $prefArr \leftarrow \langle\rangle$;
7:     for $i \leftarrow 1$ up to $size(g\bullet)$ do
8:         if $i = 1$ then
9:             $prefArr \leftarrow pref(g \bullet [1])$;
10:        else
11:            $prefArr \leftarrow prefArr \cup (prefArr[i-1] + pref(g \bullet [i]))$;
12:        end if
13:    end for
14:    $number \leftarrow random(1, prefArr[size(predArr - 1)])$;
15:    for $i \leftarrow 1$ up to $size(g\bullet)$ do
16:        if $number \leq prefArr[i]$ then
17:            return $g \bullet [i]$;
18:        end if
19:    end for
20: end function
```

Each token is associated with a timestamp which indicates when the token was added to a sequence flow. We introduce a *time* function, which returns a timestamp of a given token. Given two tokens t_1 and t_2 we say that token t_1 is earlier than token t_2 or $t_1 < t_2$, if $time(t_1) < time(t_2)$. Likewise, we say that token t_1 is later than token t_2 or $t_1 > t_2$, if $time(t_1) > time(t_2)$.

Atomic activities are the only elements whose execution times are defined. Gateways and events fire instantly. For a given atomic activity the function *StartTime* returns either a uniformly distributed value between activity's minimal and maximal execution times or allows for introduction of other distributions in a flexible manner. Activities in our approach fire without pauses in-between them. This means that the duration of the particular instance of an activity in the particular process instance defined by the starting time of the next activity instance.

See, for example, the model shown in Fig. 1. Activity “Adjust Requirement Changes” becomes enabled after the firing of both activities “Propose Changes of Internal Functional Requirements” and “Propose Changes of Internal Non-Functional Requirements”. The starting time of the gateway can be defined as the latest completion time of the previous activities. The gateway firing takes no time. Then, the starting time of the activity “Adjust Requirement Changes” is equal to the end time of the gateway firing. That is, the function *StartTime* returns for each activity a start time, which is exactly the end time of the previously fired elements.

The firing of activities is accomplished in two phases: start of firing and end of firing. At the start stage of firing a token with the earliest timestamp is removed from one of the incoming sequence flows and a token with the earliest timestamp is consumed from each incoming message flow. The start time of firing is equal to the largest timestamp among timestamps of the consumed tokens (Algorithm 5).

Then, information on the start of firing $(attr(e)(time) = StartTime(a))$ is written to the log. Next step is to log the end of firing. To do it we determine the time needed for the activity execution by invoking the *StartTime* function and adding this value to the start time. Then a token with the calculated timestamp is created for each outgoing sequence flow and information on the activity termination is added to the log.

During an execution of an enabled exclusive gateway, the earliest token is removed from the incoming sequence flows and placed to one of the outgoing sequence flows without alteration of the timestamp. When a parallel gateway fires, the earliest token is taken from each incoming sequence flow and the latest among them defines a start of the firing time. A token with this timestamp is added to each outgoing sequence flow.

6. Tool support and evaluation

6.1. Event log generator

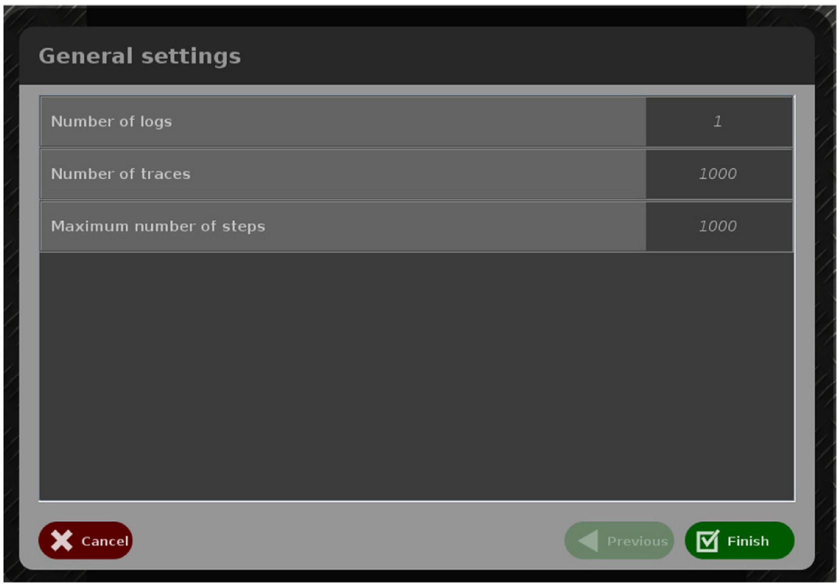
The proposed approach was implemented as a collection of plug-ins[9] for the well-known ProM tool [5]. ProM is an open-source framework for the development of process mining algorithms in a standardized environment. We used an XML-based format XES [31] to represent event logs. An event in XES is described by a corresponding activity name, timestamp, resource id, group id, and customizable properties (such as values of data variables).

The *basic log generator* implements the main generation functionality. The user is asked (see Fig. 7) to specify (1) the number of event logs to be generated, (2) the number of traces to be generated for each event log, and (3) an upper bound

[9] Available at https://sourceforge.net/projects/gena-log-generator/.

Algorithm 5 An activity start time.

```
Input: an enabled activity $a \in A$Output: start time of $a$
1: function StartTime($a$)
2:     // function token returns the earliest token from the flow
3:     // a token stored in one of the incoming sequence flows of $a$;
4:     $token_{sf}$;
5:     // a token stored in one of the incoming message flows of $a$
6:     $token_{mf}$;
7:     for all $sf$ where $sf$ is an incoming sequence flow of $a$ do
8:         if $m(sf) > 0$ then
9:             if token($sf$) $< token_{sf}$ then
10:                $token_{sf} \leftarrow$ token($sf$);
11:            end if
12:        end if
13:    end for
14:    for all $mf$ where $mf$ is an incoming message flow of $a$ do
15:        if $token(mf) > token_{mf}$ then
16:            $token_{mf} \leftarrow$ token($mf$);
17:        end if
18:    end for
19:    if $token_{sf} < token_{mf}$ then
20:        $time_a \leftarrow time(token_{mf})$;
21:    else
22:        $time_a \leftarrow time(token_{sf})$;
23:    end if
24:    return $time_a$;
25: end function
```


General settings

Number of logs	1
Number of traces	1000
Maximum number of steps	1000

Fig. 7. General settings.

for the length of a trace (see Section 5.1). Using the *extended BPMN log generator* the user can easily configure the behavior of an existing model via configuration of exclusive gateways. It is also possible to assign special preferences to gateways as described in Section 5. The *time-aware log generator* is responsible for the generation of event logs with timestamps. The *data-aware generator* allows for using models with explicit data-flows.

The actual implementation of data objects extends their definition given in Section 4. To make simulation more flexible and configurable user-written Python scripts are supported. There are three types of nodes which can be enriched with scripts: (1) data objects, (2) activities including nested processes, and (3) exclusive choice gateways. Scripts can be used to initialize data objects and specify actions performed with them during the node firing.

Each node (of the three mentioned types) can be associated with of most one script. If a data object is not associated with any initializing script, its default value equals to an empty string. Otherwise, a script is called when a generation of a trace starts.

Now let us demonstrate how to generate an event log for the model of Architectural Changes Process from Section 3 (see Fig. 1) using our tool. The maximal number of firings per trace was set to 200. We decided to use data-aware log gener-

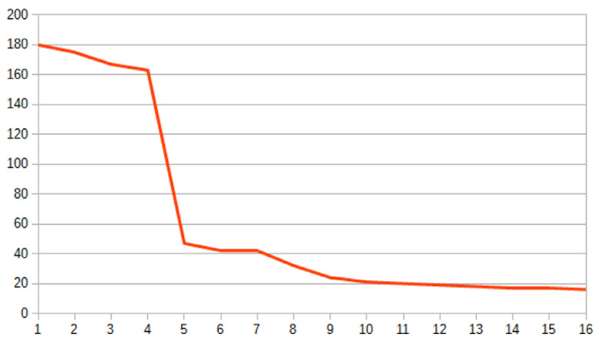


Fig. 8. Distribution of traces.

Table 1
A one-trace fragment of example generated event log.

Trace	Event	Activity name	Group	Role	Timestamp / Data variables
21	1	New requirement formulated	Customer	–	2016-02-24T22:02:24.455+03:00 [-;“Functional Reqs”;-;-;-]
21	2	Wait for proposal	Customer	–	2016-02-24T23:08:16.455+03:00 [-;“Functional Reqs”;-;-;-]
21	3	New requirements request	Arch.team	Analyst	2016-02-25T00:01:00.455+03:00 [-;“Functional Reqs”;-;-;-]
21	4	Propose changes of internal non-func reqs	Arch.team	Analyst	2016-02-25T01:49:28.455+03:00 [-;“Functional Reqs”;-;-;-]
21	5	Propose changes of internal func reqs	Arch.team	Analyst	2016-02-25T01:55:24.455+03:00 [-;“Functional Reqs”;-;-;-]
21	6	Adjust requirement changes	Arch.team	Analyst	2016-02-25T02:58:31.455+03:00 [-;“Functional Reqs”;-;-;-]
21	7	Check requirements changes	Arch.team	Architect	2016-02-25T04:45:48.455+03:00 [-;“Functional Reqs”;-;-;-]
21	8	Propose architectural changes	Arch.team	Architect	2016-02-25T05:48:38.455+03:00 [-;“Functional Reqs”;203.37;-;-]
21	9	Calculate cost	Arch.team	Analyst	2016-02-25T07:01:46.455+03:00 [-;“Func.Reqs”;203.37;426.75;-]
21	10	Send proposal	Arch.team	Analyst	2016-02-25T08:04:16.455+03:00 [-;“Func.Reqs”;203.37;426.75;-]
21	11	Receive proposal	Customer	–	2016-02-25T09:33:47.455+03:00 [14.1;“Func.Reqs”;203.37;426.75;-]
21	12	Reject proposal	Customer	–	2016-02-25T11:25:35.455+03:00 [14.1;“Func.Reqs”;203.37;426.75;-]
21	13	Analyze results	Customer	–	2016-02-25T12:28:49.455+03:00 [14.1;“Func.Reqs”;203.37;426.75;-]
21	14	Proposal rejected	Arch.team	Analyst	2016-02-25T12:46:36.455+03:00 [14.1;“Func.Reqs”;203.37;426.75;-]

ation with time. The generated sample event log contains 1000 traces, 13,119 events, and 21 activity classes (the same number as activities in the model shown in Fig. 1). The model contains five data objects. Thus, each event in the log will contain values for all five data variables: `[Cost for customer: Float; Functional requirements: String; Duration: Float; Cost: Float; Total budget: Float]`. There are 16 different case variants (sets of traces presenting the same sequence of event names) in the generated log. This number is a property of a particular model used for the log generation. Model shown in Fig. 1 specifies three general cases: one successful case, when all stakeholders accept proposals (with 17 events), and two rejection cases (with 12 and 14 events). Other cases lead to deadlocks. The number of actual case variants in the log is larger because of choices and interleaving of concurrent events. For example, in the process functional requirements can be proposed before or after of non-functional requirements.

Fig. 8 shows the distribution of number of traces by case variants. One can see, that there are three groups of cases. Approximately 700 traces are from the four most popular cases: 1, 2, 3, 4. Several cases form the second group of less popular cases (numbered 5, 6, and 7). The over variants (case types 8 to 16) are even less frequent. Obviously, short traces are the most popular. The longer the trace, the more variations it allows. Thus, the structure of a model together with the generation settings influences the character of the distribution. One can specify preferences, which will lead to a single-case event log, by disabling all branches except the desired ones. The distribution shown in Fig. 8 is typical for the generation with random choices on all gateways. Table 1 contains one-trace fragment from the generated event log.

Table 2
Simulation statistics for 10 different models selected from Signavio collection.

Model iD	Number of elements (Events)	Activities/ Gateways(XOR+AND)/ Flows	Pools/ Lanes	Trace variants	Total events (Classes)	Trace length		
						Min	Mean	Max
8914774_rev1	11 (2)	5 / 4+0 / 12	0 / 0	3	2511 (5)	2	3	3
100193728_rev3	16 (2)	10 / 2+2 / 19	1 / 4	1000	30,930 (10)	21	31	64
1073989552_rev1	17 (4)	9 / 2+2 / 17	0 / 0	5	5072 (9)	3	5	7
1754711371_rev1	18 (2)	10 / 4+2 / 20	1 / 2	6	5905 (10)	4	6	9
186202353_rev1	30 (2)	16 / 4+8 / 35	0 / 0	657	18,024 (16)	15	18	57
1280191109_rev7	28 (3)	22 / 3+0 / 30	0 / 0	1000	896,988 (22)	890	897	904
604624904_rev1	27 (2)	17 / 0+8 / 15	1 / 4	36	17,000 (17)	17	17	17
1391700380_rev1	18 (2)	10 / 4+2 / 20	0 / 0	6	5962 (10)	4	6	9
1849720729_rev3	12 (4)	6 / 2+0 / 11	1 / 3	3	3505 (6)	3	4	4
2012957934_rev3	21 (3)	13 / 2+3 / 23	0 / 0	28	7228 (13)	1	7	35

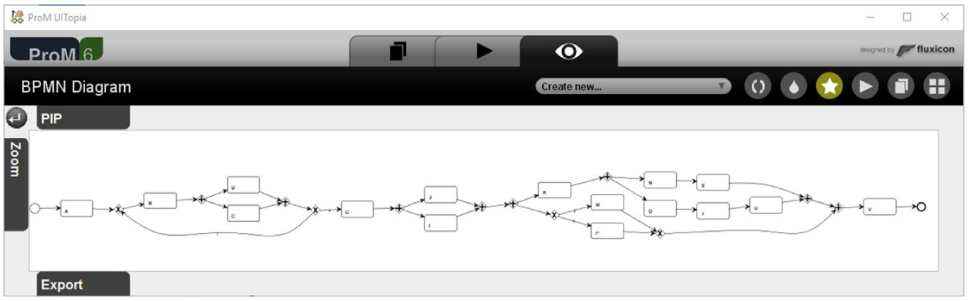


Fig. 9. Initial model.

6.2. Evaluation

In order to test robustness and effectiveness of the approach, the presented tool has been tested on models selected from the Signavio BPMN Reference Models Collection[10]. This collection contains several thousands (more than 4 700) of real-life process models made by experts. These models were collected by Signavio from many of application areas and are usually used to test analysis algorithms in the BPM field.

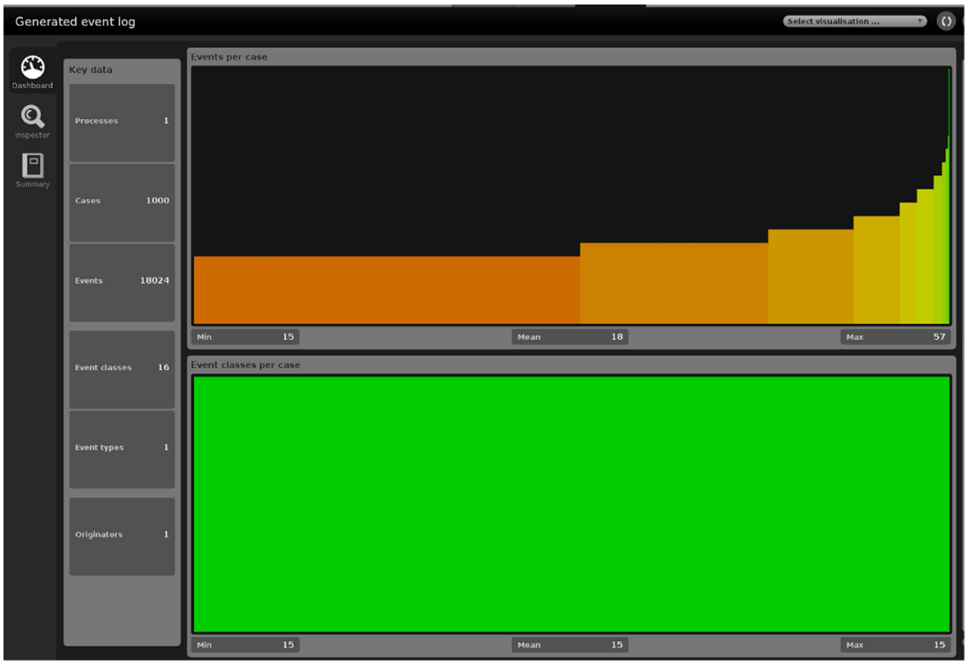
Approximately 3000 of models satisfy the restrictions of the formal framework described in Section 4. The most popular construct, which is not supported in our framework, is a *message event*. In general, these events are a type of syntactic sugar, they add no essential aspects for the modeling. Another non-supported type of elements is *timer event*. However, we plan to support these elements in the next version of the tool.

The event log generator works correctly and robustly on 956 of satisfying models. For the other models the tool fails to return traces after 1000 attempts. These models can not terminate because of deadlocks (process executions leading to the states, in which no node is enabled) or livelocks (executions inevitably ending up in a loop of repeating tasks without possibility to reach the end state). For each model the test script generated an event log of 500 traces using our tool. The whole test procedure (for all models from the Signavio Collection) took 646,065 ms (10 min approximately) on a typical desktop computer. The total number of generated traces is 478,000. During the generation 10,520 deadlocks and 198,663 livelocks were identified. Our tool successfully simulated models from Signavio Collection, which satisfy the restrictions of the formal framework, and contain no deadlocks or livelocks.

Table 2 shows generation statistics for 10 different models, which were randomly selected from Signavio Collection. 1000 traces were generated for each model. The reader can see the relationship between model structure (number of activities, gateways, lanes) and the variability of characteristics of generated event logs. In Fig. 9 the `186202353_rev1` model is shown. Fig. 10 shows the view with characteristics of the event log, which was generated for this model.

In the worst case complexity for the log generation can be estimated as $C = O(N_L \cdot N_{tr} \cdot S_f \cdot (|SF| + |MF| + |N| + |DO| + E_v))$, where N_L is the number of event logs, N_{tr} is the number of traces, S_f is a maximum number of firings within a trace generation, *SF*, *MF*, *N*, and *DO* are the total numbers of sequence flows, message flows, sequence nodes, and data objects respectively, E_v is a constant, indicating the number of operations needed to generate an event (depends on the generation type). Before a single firing all sequence and message flows are checked for the presence of tokens, after that activated nodes are determined, then one of them is selected and fired, during its firing novel tokens are produced, the values of data variables are changed and not more than E_v log writing operations are performed.

[10] Signavio models collection: http://www.signavio.com/reference-models/.



Processes 1
Cases 1000
Events 18024
Event classes 16
Event types 1
Originators 1

Min 15 Mean 18 Max 57

Min 15 Mean 15 Max 15

Fig. 10. Generated event log.

7. Conclusion

This paper proposes an approach for the generation of event logs for high-level process models. Formal semantics for the core part of the BPMN standard, including flat control flow, hierarchy, data perspective, and participants, are defined and supported. On the basis of the introduced semantics, a tool for the automated event log generation has been implemented within the ProM Framework. Furthermore, the generation is enriched with capabilities of modifying the behavior of exclusive gateways and simulating models with time.

The approach has been tested using hundreds of models from the Signavio BPMN Reference Models Collection. The results of this evaluation showed that the approach allows the generation of event logs for a significant portion of models from this collection. Moreover, the tool has shown its effectiveness, efficiency and robustness. The entire testing collection can be processed in less than half an our. The tool is resistant to unsound models that get stuck in deadlocks or livelocks. Experimentations also show the need to expand the number of BPMN elements supported by the tool. Several types of events are not yet supported by our tool but are present in the model collection. This is especially true for message events. Nevertheless, the core part of BPMN model is supported by our approach.

With the help of this tool, researchers and BPM specialists can evaluate process mining algorithms using sets of event logs with desired characteristics generated for fine-tuned models. A possible extension of the approach can be to incorporate more elements from the BPMN 2.0 standard. The clear identification of dependencies between specific model parameters and statistical characteristics of corresponding event logs may also be an interesting area for future research.

References

[1] OMG, Business Process Model and Notation (BPMN), Object Management Group, 2013. (formal/2013-12-09).
[2] A.A. Kalenkova, M. de Leoni, W.M.P. van der Aalst, Discovering, analyzing and enhancing BPMN models using ProM, in: Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014, CEUR Workshop Proceedings, vol. 1295, CEUR-WS.org, 2014, p. 36.
[3] A.A. Kalenkova, W.M.P. van der Aalst, I.A. Lomazova, V.A. Rubin, Process mining using BPMN: relating event logs and process models, Softw. Syst. Model. (2015) 1–30.
[4] R. Conforti, M. Dumas, L. Garca-Bauelos, M. La Rosa, Beyond tasks and gateways: discovering BPMN models with subprocesses, boundary events and activity markers, in: Business Process Management, in: Lecture Notes in Computer Science, vol. 8659, Springer International Publishing, 2014, pp. 101–117.
[5] H. Verbeek, J. Buijs, B. Dongen, W. Aalst, ProM 6: the process mining toolkit, in: Proc. of BPM Demonstration Track 2010, CEUR Workshop Proceedings, vol. 615, 2010, pp. 34–39.
[6] W.M.P. van der Aalst, Process Mining - Data Science in Action, second ed., Springer, 2016.
[7] A.K.A.d. Medeiros, C.W. Günther, Process mining: using CPN tools to create test logs for mining algorithms, in: K. Jensen (Ed.), Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005), DAIMI, vol. 576, University of Aarhus, Aarhus, Denmark, 2005, pp. 177–190.
[8] M. Zäuram, Business Process Simulation Using Coloured Petri Nets, University of Tartu, Estonia, 2010 Master's thesis.

[9] M. Ramadan, H. Elmongui, R. Hassan, BPMN formalisation using coloured petri nets, in: The 2nd GSTF Annual International Conference on Software Engineering & Applications (SEA'11), 2011.
[10] B.P. Zeigler, Hierarchical, modular discrete-event modelling in an object-oriented environment, Simulation 49 (5) (1987) 219–230, doi:10.1177/003754978704900506.
[11] G. Wainer, DEVS tools, Accessed: 2016-11-03,.
[12] D. Cetinkaya, A. Verbraeck, M.D. Seck, Model transformation from BPMN to DEVS in the MDD4MS framework, in: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, TMS/DEVS '12, Society for Computer Simulation International, San Diego, CA, USA, 2012, pp. 28:1–28:6.
[13] H. Bazoun, Y. Bouanan, G. Zacharewicz, Y. Ducq, H. Boye, Business process simulation: transformation of BPMN 2.0 to DEVS models (wip), in: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative, DEVS '14, Society for Computer Simulation International, San Diego, CA, USA, 2014, pp. 20:1–20:7.
[14] S. Boukelkoul, R. Maamri, Optimal model transformation of BPMN to DEVS, in: 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), 2015, pp. 1–8, doi:10.1109/AICCSA.2015.7507115.
[15] T. Stocker, R. Accorsi, SecSy: security-aware Ssynthesis of process event logs, in: Proceedings of the 5th International Workshop on Enterprise Modelling and Information Systems Architectures, 2013. St. Gallen, Switzerland.
[16] A. Burattin, A. Sperduti, PLG: a framework for the generation of business process models and their execution logs, in: BPM 2010 Workshops, Proceedings of the Sixth Workshop on Business Process Intelligence (BPI2010), Lecture Notes in Business Information Processing, vol. 66, Springer-Verlag, Berlin, 2011.
[17] A. Burattin, PLG2: multiperspective processes randomization and simulation for online and offline settings, CoRR abs/1506.08415 (2015).
[18] V.M. Kataeva, A.A. Kalenkova, Applying graph grammars for the generation of process models and their lsogs, in: Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014), 2014, pp. 83–87.
[19] A. Rozinat, R.S. Mans, M. Song, W.M.P. van der Aalst, Discovering simulation models, Inf. Syst. 34 (3) (2009a) 305–327.
[20] A. Rozinat, M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, C.J. Fidge, Workflow simulation for operational decision support, Data Knowl. Eng. 68 (9) (2009b) 834–850.
[21] K.M. van Hee, M.L. Rosa, Z. Liu, N. Sidorova, Discovering characteristics of stochastic collections of process models, in: BPM, Lecture Notes in Computer Science, vol. 6896, Springer, 2011a, pp. 298–312.
[22] K.M. van Hee, Z. Liu, N. Sidorova, Is my event log complete? - A probabilistic approach to process mining, in: RCIS, IEEE, 2011b, pp. 1–7.
[23] R.M. Dijkman, M.L. Rosa, H.A. Reijers, Managing large collections of business process models - current techniques and challenges, Comput. Ind. 63 (2) (2012) 91–97.
[24] I.S. Shugurov, A.A. Mitsyuk, Generation of a set of event logs with noise, in: Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014), 2014, pp. 88–95.
[25] M. zur Muehlen, J. Recker, How much language is enough? Theoretical and practical use of the business process modeling notation, in: Seminal Contributions to Information Systems Engineering, Springer, 2013, pp. 429–443.
[26] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.
[27] A. Kheldoun, K. Barkaoui, M. Ioualalen, Specification and verification of complex business processes - a high-level petri net-based approach, in: BPM, Lecture Notes in Computer Science, vol. 9253, Springer, 2015, pp. 55–71.
[28] J. Ye, S. Sun, W. Song, L. Wen, Formal semantics of BPMN process models using YAWL, in: Intelligent Information Technology Application, 2008. IITA '08. Second International Symposium on, vol. 2, 2008, pp. 70–74.
[29] A. Kalenkova, I.A. Lomazova, Discovery of cancellation regions within process mining techniques, Fundam. Inform. 133 (2–3) (2014) 197–209.
[30] W.M.P. van der Aalst, A.A. Kalenkova, V.A. Rubin, H. Verbeek, Process discovery using localized events, in: R. Devillers, A. Valmari (Eds.), Application and Theory of Petri Nets and Concurrency: 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21–26, 2015, Proceedings, Springer International Publishing, Cham, 2015, pp. 287–308.
[31] IEEE Task Force on Process Mining, XES Standard Definition, 2013. (www.xes-standard.org).