Generating Event Logs through the Simulation of Declare Models

Claudio Di Ciccio[1], Mario Luca Bernardi[2], Marta Cimitile[3], and Fabrizio Maria Maggi[4]

[1] Vienna University of Economics and Business, Austria
claudio.di.ciccio@wu.ac.at
[2] University of Sannio, Italy
mlbernar@unisannio.it
[3] Unitelma Sapienza University, Italy
marta.cimitile@unitelma.it
[4] University of Tartu, Estonia
f.m.maggi@ut.ee

Abstract. In the process mining field, several techniques have been developed during the last years, for the discovery of declarative process models from event logs. This type of models describes processes on the basis of temporal constraints. Every behavior that does not violate such constraints is allowed, and such characteristic has proven to be suitable for representing highly flexible processes. One way to test a process discovery technique is to generate an event log by simulating a process model, and then verify that the process discovered from such a log matches the original one. For this reason, a tool for generating event logs starting from declarative process models becomes vital for the evaluation of declarative process discovery techniques. In this paper, we present an approach for the automated generation of event logs, starting from process models that are based on Declare, one of the most used declarative modeling languages in the process mining literature. Our framework bases upon the translation of Declare constraints into regular expressions and on the utilization of Finite State Automata for the simulation. An evaluation of the implemented tool is presented, showing its effectiveness in both the generation of new logs and the replication of the behavior of existing ones. The presented evaluation also shows the capability of the tool of generating very large logs in a reasonably small amount of time, and its integration with state-of-the-art Declare modeling and discovery tools.

Keywords: Declare, Regular Expressions, Declarative Process Models, Process Simulation, Log Generation

1 Introduction

Process mining is a rising research discipline allowing for the analysis of business processes starting from event logs. XES (eXtensible Event Stream) [1] has been

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

recently developed as the standard for storing, exchanging and analyzing event logs. In this standard, each event refers to an activity (i.e., a well-defined step in some process) [2, 3] and is related to a particular case (i.e., a process instance). The events belonging to a case are ordered and can be seen as one execution of the process (often referred to as a trace of events). Event logs may store additional information about events such as the resource (i.e., person or device) executing or initiating the activity, the timestamp of the event, or data elements recorded with the event.

One of the main branches of process mining is the automated discovery of process models from event logs. The main idea of process discovery is to extract knowledge from logs concerning control flow, data, organizational and social structures. Therefore, testing and evaluation of process discovery techniques and tools require the availability of event logs. There are several real life logs publicly available that can be used for this purpose [4, 5]. However, these logs usually contain imperfections and have some missing information that can alter the evaluation of the discovery algorithms (e.g., they can be incomplete and/or contain noise). For this reason, a common approach adopted for testing process discovery algorithms is based on the use of synthetic logs created via simulation. Simulations can produce event logs with different predefined characteristics and allow the researchers to have more control on the experimental settings to fine tune the developed algorithms.

Starting from these needs, several model simulators and log generators have been developed and are available in the literature [6, 7, 8, 9]. However, all these tools generate synthetic logs through the simulation of a procedural process model. This makes them not suitable for the evaluation of process discovery techniques based on declarative process models. Such techniques have recently attracted the attention of the process mining community and are useful to mine processes working in dynamic environments [10, 11, 12, 13, 14, 15, 16]. Indeed, differently from procedural process models that work in a closed world assumption and explicitly specify all the allowed behaviors, declarative models are open. Therefore, they enjoy flexibility and are more suitable to describe highly variable behaviors in a compact way.

To test process discovery techniques based on declarative models, tools for the generation of event logs based on the simulation of declarative models are needed and, to the best of our knowledge, they are not available in the literature. To close this gap, in this paper, we present a tool for log generation based on Declare models [17]. This model simulator is based on the translation of Declare constraints into regular expressions and the utilization of Finite State Automata for the simulation of declarative processes. The tool allows the user to generate logs with predefined characteristics (e.g., number and length of the process instances), which is compliant with a given Declare model.

The paper is structured as follows. In Section 2, some background concepts are discussed. Section 3 describes the proposed approach. Section 4 reports the experimental setup and the results of the experiments. Section 5 discusses the

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

related work. Section 6 contains conclusive remarks and briefly presents future work.

2 Background

In this section, we describe some background elements of our proposed research, i.e., the concepts of event log, Declare-based modeling of processes, and essential theoretical notions about regular expressions and Finite State Automata.

2.1 Event Logs

A basic functionality of the core component of Business Process Management Systems (BPMSs) is the recording, in the so-called *event logs* [18], of reporting information during the execution of a workflow [19]. An event log is a structured text file documenting the executions of a single process. Each event log indeed contains a collection of *traces*, each representing the enactment of a unique case (a process instance). Traces are in turn sequences of *events*, i.e., single data entries related to the carry-out of an activity, within the process instance evolution. In 2010, the IEEE Task Force on Process Mining has adopted XES (eXtensible Event Stream) [1] as the standard for storing, exchanging and analyzing event logs. Event logs are typically stored during the Business Process Management System (BPMS)-aided execution of a process. Therefore, logs tend to respect the process model that the BPMS's execution engine loads to coordinate the workflow. In the next section, we introduce Declare, a process modeling notation that is alternative to the older well-established procedural languages, such as Petri nets [20], Workflow Nets [21], YAWL [22] and BPMN [19].

2.2 Declare

In this work, the process models are meant to be defined using Declare, a declarative process modeling language introduced by Pesic and van der Aalst in [24]. Declare is qualified as "declarative" because it does not explicitly specify every possible sequence of activities leading from the start to the end of a process execution. Instead, it bases models upon a set of constraints, which must hold true during the enactment. All behaviors that respect the constraints are thus allowed. Constraints are meant to be exerted on sets of activities and mainly pertain to their temporal ordering. In particular, Declare specifies an extensible set of standard templates (see Table 1) that a process analyst can use to model a process. Constraints are concrete instantiations of templates. The adoption of templates makes the model comprehension independent of the logic-based formalization. Indeed, analysts can work with the graphical representation of templates while the underlying formulas remain hidden. Graphically, a Declare process model is a diagram, where activities are presented as nodes (labeled rectangles), and constraints as arcs between activities.

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

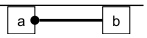
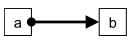
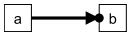
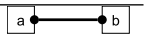
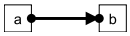
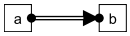
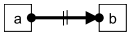
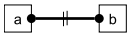
	Template	Regular Expression	Notation
Existence	*Participation*(a)	`[^a]*(a[^a]*)[^a]*`	
	AtMostOne(a)	`[^a]*(a)?[^a]*`	
	Init(a)	`a.*`	
	End(a)	`.*a`	
Relation	*RespondedExistence*(a, b)	`[^a]*((a.*b.*)|(b.*a.*))*[^a]*`	
	Response(a, b)	`[^a]*(a.*b)*[^a]*`	
	AlternateResponse(a, b)	`[^a]*(a[^a]*b[^a]*)*[^a]*`	
	ChainResponse(a, b)	`[^a]*(ab[^a]*)*[^a]*`	
	Precedence(a, b)	`[^b]*(a.*b)*[^b]*`	
	AlternatePrecedence(a, b)	`[^b]*(a[^b]*b[^b]*)*[^b]*`	
	ChainPrecedence(a, b)	`[^b]*(ab[^b]*)*[^b]*`	
Coupling Relation	*CoExistence*(a, b)	`[^ab]*((a.*b.*)|(b.*a.*))*[^ab]*`	
	Succession(a, b)	`[^ab]*(a.*b)*[^ab]*`	
	AlternateSuccession(a, b)	`[^ab]*(a[^ab]*b[^ab]*)*[^ab]*`	
	ChainSuccession(a, b)	`[^ab]*(ab[^ab]*)*[^ab]*`	
Negative Relation	*NotChainSuccession*(a, b)	`[^a]*(aa*[^ab][^a]*)*([^a]*|a)`	
	NotSuccession(a, b)	`[^a]*(a[^b]*)*[^ab]*`	
	NotCoExistence(a, b)	`[^ab]*((a[^b]*)|(b[^a]*))?`	

Table 1: Semantics of Declare templates as POSIX regular expressions [23].

Compared with procedural approaches, Declare models are more suitable to describe processes working in unstable environments and characterized by many exceptional behaviors. Since all what is not explicitly specified is allowed, few constraints can specify many possible behaviors at once.

Declare templates can be divided into two main groups: *existence templates* and *relation templates*. The former is a set of unary templates. They can be expressed as predicates over one variable. The latter comprises rules that are imposed on target activities, when activation tasks occur. Relation templates thus correspond to binary predicates over two variables. Starting from the first row of Table 1, *Participation*(a) is an existence template, which requires the execution of a at least once in every process instance. *AtMostOne*(a) is its dual, as it details that a is not executed more than once in a process instance. *Init*(a) and *End*(a) specify that a occurs in every case as the first and the last activity, respectively. *RespondedExistence*(a, b) is a relation template imposing that if a is performed at least once during the process execution, b must occur at least once as well, either in the future or in the past, with respect to a. *Response*(a, b) adds to *RespondedExistence*(a, b) the condition that b must occur eventually *after* a. *AlternateResponse*(a, b) adds to *Response*(a, b) the condition that no other a's

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

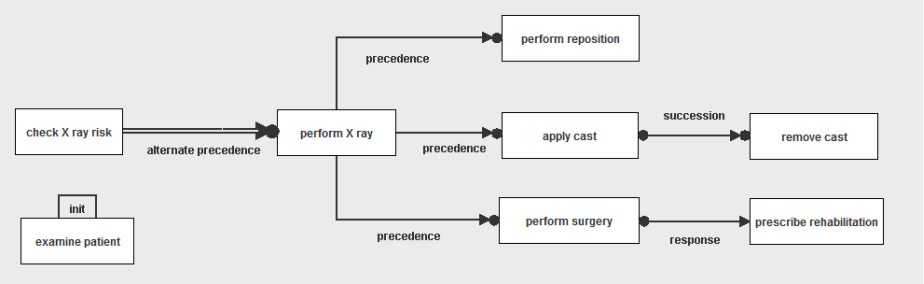


Fig. 1: The Declare model for a fracture treatment process.

occur between an execution of a and a subsequent b. *ChainResponse*(a, b) is even stronger and specifies that whenever a occurs, b must occur immediately after. *Precedence*(a, b) specifies that a must occur *before* b. *AlternatePrecedence*(a, b) adds to *Precedence*(a, b) the condition that no other b's occur between an execution of b and a precedent a. *ChainPrecedence*(a, b) specifies that whenever b occurs, a must occur immediately before.

Two specializations of the relation templates are *coupling relation templates* and *negative relation templates*. In the first group there are templates where both the constrained activities are together activation and target. For instance, *CoExistence*(a, b) is a coupling relation template requiring that if a is executed, then b must be performed as well, and vice-versa. In the second group, the occurrence of one activity excludes the occurrence of the other. For instance, *NotCoExistence*(a, b) is a negative relation template requiring that if a is executed, then b cannot be performed in the same trace, and vice-versa.

An example of Declare process model (fracture treatment) is depicted in Fig. 1. The process comprises activities *examine patient*, *check X ray risk*, *perform X ray*, *perform reposition*, *apply cast*, *remove cast*, *perform surgey*, and *prescribe rehabilitation*. Its behavior is specified by the following constraints C_1–C_7:

C_1 *Init*(*examine patient*)
C_2 *AlternatePrecedence*(*check X ray risk*, *perform X ray*)
C_3 *Precedence*(*perform X ray*, *perform reposition*)
C_4 *Precedence*(*perform X ray*, *apply cast*)
C_5 *Succession*(*apply cast*, *remove cast*)
C_6 *Precedence*(*perform X ray*, *perform surgery*)
C_7 *Response*(*perform surgery*, *prescribe rehabilitation*)

According to these constraints, every process instance starts with activity *examine patient*. Moreover, if activity *perform X ray* occurs, then *check X ray risk* must be carried out before it, without other occurrences of *perform X ray* in between. Activities *perform reposition*, *apply cast* and *perform surgery* require that *perform X ray* occurs before they are executed. If *perform surgery* occurs, then *prescribe rehabilitation* occurs eventually after it. Finally, after every exe-

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

cution of *apply cast*, eventually *remove cast* occurs and, vice-versa, before every occurrence of *remove cast*, *apply cast* must be carried out.

Declare templates semantics have been expressed in the literature as formulations of several formal languages: as Linear Temporal Logic over Finite Traces (LTL_f) formulas [25], as shown in [26]; in the form of $\mathcal{S}$CIFF integrity constraints [27], as exploited in [28]; as First Order Logic (FOL) formulas interpreted over finite traces, as described in [13, 25]. In this work, we use regular expressions (REs), as described in [23]. Table 1 reports the translation of Declare constraints into regular expressions(REs).

2.3 Regular Expressions

Regular expressions are a formal notation to compactly express finite sequences of characters, a.k.a. matching patterns. The syntax of REs consists of any juxtaposition of characters of a given alphabet, optionally grouped by enclosing parentheses `(` and `)`, to which the following well-known operators can be applied: the binary alternation `|` and concatenation, and the unary Kleene star `*`. Thus, the regular expression `a(bc)*d|e` identifies any string starting with `a`, followed by any number of repetitions of the pattern (sub-string) `bc` (optionally, none), and closed by either `d` or `e`, such as `ad`, `abcd`, `abcbce` and `ae`. Table 1 adopts the POSIX standard for the following additional shortcut notations: *(i)* `.` and `[^`x`]` respectively denote any character, or any character but x, *(ii)* the `+` and `?` operators respectively match from one to any, and none to one occurrences of the preceding pattern. In the reminder of this paper, we will also make use of *(iii)* the parametric quantifier $\{,m\}$, with m integer higher than 0, which specifies the maximum number of repetitions of the preceding pattern, and *(iv)* the parametric quantifier $\{n,\}$, with n integer higher than or equal to 0, which specifies the minimum number of repetitions of the preceding pattern. We recall here that *(i)* REs are closed under the conjuction operation & [29], and *(ii)* the expressive power of REs completely covers regular languages, thus *(iii)* for every RE, a corresponding determinstc Finite State Automaton (FSA) exists, accepting all and only the matching strings.

2.4 Finite State Automata

A determinstc FSA is a labeled transition system $\mathcal{A} = \langle A, S, \delta, s_0, S_\text{f} \rangle$ defined over states S and an alphabet A, having $\delta : S \times A \to S$ as transition function, i.e., a function that, given a starting state and a character, returns the target state (if defined). $s_0 \in S$ is the initial state of $\mathcal{A}$, and $S_\text{f} \subseteq S$ is the non-empty set of its accepting states ($S_\text{f} \neq \emptyset$). For the sake of simplicity, we will omit the qualification "deterministic" in the remainder of this paper. A finite path π of length n over $\mathcal{A}$ is a sequence $\pi = \langle \pi^1, \ldots, \pi^n \rangle$ of tuples $\pi^i = \langle s^{i-1}, \sigma^i, s^i \rangle \in \delta$, for which the following conditions hold true: *(i)* π^1, the first tuple, is such that $s^0 = s_0$ (it starts from the initial state of $\mathcal{A}$), and *(ii)* the starting state of π^i is the target state of π^{i-1}: $\pi = \langle \langle s^0, \sigma^1, s^1 \rangle, \langle s^1, \sigma^2, s^2 \rangle, \ldots \langle s^{n-1}, \sigma^n, s^n \rangle \rangle$.

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

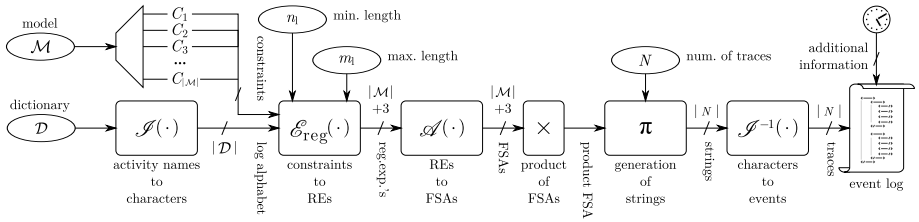


Fig. 2: The log-generation framework.

A finite string of length $n \geqslant 1$, i.e., a concatenation $t = t_1 \dots t_n$ of characters $t_i \in A$ is accepted by $\mathcal{A}$ if a path π of length n is defined over $\mathcal{A}$ and is such that *(i)* for every $i \in [1, n]$, $\pi^i = \langle s^{i-1}, t_i, s^i \rangle$, and *(ii)* $\pi^n = \langle s^{i-1}, t_n, s^n \rangle$ is s.t. $s^n \in S_f$.

FSAs are closed under the product operation $\times$. A product of two FSAs accepts the intersection of languages (sets of accepted strings) accepted by each operand. The product of FSAs is an isomorphism for the conjunction of RE, i.e., the product of two FSAs respectively corresponding to two REs is equivalent to the FSA that derives from the conjunction of the two REs.

3 Approach

Fig. 2 sketches the modular framework upon which our approach is based. The output is an event log L, synthesized on the basis of a Declare process model that must regulate the composition of the traces. The input indeed consists of *(i)* the set of activity names of the process, henceforth activity dictionary $\mathcal{D}$, *(ii)* a Declare model $\mathcal{M}$, i.e., a set of constraints $C_1, \dots, C_{|\mathcal{M}|}$ expressed on the activities of the process, *(iii)* the minimum and maximum number of events per trace, respectively n_l and m_l, and *(iv)* the number N of traces that the output event log must contain.

The overall approach goes through six consecutive steps, as listed below. The idea is to create a Finite State Automaton that accepts all and only those traces complying with the conditions imposed by the user. The event log will result in a collection of traces, generated by running paths along the automaton, from its initial state, to an accepting one.

From Activity Names to Characters. The first step is the mapping of process activity names to single terminal characters, henceforth *process alphabet* Σ, by means of the function $\mathscr{I} : \mathcal{D} \to \Sigma$. $\mathscr{I}$ is bijective, i.e., every activity maps to a distinct character, and all characters can be referenced to the related activity. Therefore, it admits an inverse function $\mathscr{I}^{-1} : \Sigma \to \mathcal{D}$. In the example of Fig. 1, activities of the process are *apply cast*, *check X ray risk*, *examine patient*, *perform X ray*, *perform reposition*, *perform surgery*, *prescribe rehabilitation*, and *remove cast* in $\mathcal{D}$, respectively mapped by $\mathscr{I}$ to a, b, c, d, e, f, g, and h in Σ. Thus, e.g., $\mathscr{I}(\textit{perform surgery}) = \mathsf{f}$, and $\mathscr{I}^{-1}(\mathsf{g}) = \textit{prescribe rehabilitation}$.

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

From Constraints to Regular Expressions. Every constraint is thereafter translated into the corresponding RE during the second step, as per Table 1 [23]. The translation function is henceforth indicated with $\mathscr{E}_{\text{reg}}$. In the example, $\mathscr{E}_{\text{reg}}(\mathit{Response}(\mathtt{f},\mathtt{g})) = \texttt{[^f]*(f.*g)*[^f]*}$, being $\mathtt{f} = \mathscr{I}(\textit{perform surgery})$ and $\mathtt{g} = \mathscr{I}(\textit{prescribe rehabilitation})$.

For the generation of logs, an additional regular expression is considered, specifying that accepted strings can only comprise those characters that belong to the process alphabet [30]: $\left[\sigma_1\sigma_2\ldots\sigma_{|\Sigma|}\right]*$, for all $\sigma_{\mathtt{i}} \in \Sigma, \mathtt{i} \in [1,|\Sigma|]$. In the example, such RE is $[\texttt{abcdefgh}]*$. By means of regular expressions, we also specify *(a)* the minimum, and *(b)* the maximum length of traces. Given the user-defined parameters n_{l} and m_{l}, such REs are *(a)* $\texttt{.*\{}n_{\text{l}}\texttt{,\}}$, and *(b)* $\texttt{.*\{,}m_{\text{l}}\texttt{\}}$.

Therefore, the specification of the traces for the fracture treatment process $\mathcal{M}$ consisting of constraints C_1–C_7 (Fig. 1, Section 2.2) defined over activities in $\mathcal{D}$, where the length of traces ranges between $n_{\text{l}} = 3$ and $m_{\text{l}} = 20$, would result in the following list of REs, R_1–R_{10}:

R_1 $\texttt{.*c} = \mathscr{E}_{\text{reg}}(C_1)$
R_2 $\texttt{[^d]*(b[^d]*d[^d]*)*[^d]*} = \mathscr{E}_{\text{reg}}(C_2)$
R_3 $\texttt{[^e]*(d.*e)*[^e]*} = \mathscr{E}_{\text{reg}}(C_3)$
R_4 $\texttt{[^a]*(d.*a)*[^a]*} = \mathscr{E}_{\text{reg}}(C_4)$
R_5 $\texttt{[^ah]*(a.*h)*[^ah]*} = \mathscr{E}_{\text{reg}}(C_5)$
R_6 $\texttt{[^f]*(d.*f)*[^f]*} = \mathscr{E}_{\text{reg}}(C_6)$
R_7 $\texttt{[^f]*(f.*[g])*[^f]*} = \mathscr{E}_{\text{reg}}(C_7)$
R_8 $\texttt{[abcdefgh]*}$
R_9 $\texttt{.*\{3,\}}$
R_{10} $\texttt{.*\{,20\}}$

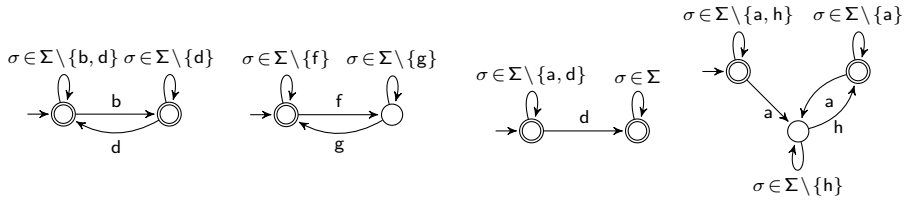
Out of these regular expressions, R_1–R_7 are the respective translation of constraints C_1–C_7 given in Section 2.2. R_8 defines the characters that are admissible for the strings, whilst R_9 and R_{10} limit their length. The output of this phase is thus a set of $|\mathcal{M}| + 3$ REs.

From Regular Expressions to Finite State Automata. For each RE, a FSA accepting all and only the matching strings is derived, by means of function $\mathscr{A}$. Fig. 3 shows the FSAs deriving from those regular expressions that express the relation constraint templates in our example model (*AlternatePrecedence*, *Response*, *Precedence* and *Succession*). In particular, $\mathcal{A}_2 = \mathscr{A}(R_2)$ is depicted in Fig. 3a, $\mathcal{A}_7 = \mathscr{A}(R_7)$ in Fig. 3b, $\mathcal{A}_4 = \mathscr{A}(R_4)$ in Fig. 3c, and $\mathcal{A}_5 = \mathscr{A}(R_5)$ in Fig. 3d, respectively referring to constraints C_2, C_7, C_4, and C_5. Hence, the outcome of this phase is a set of $|\mathcal{M}| + 3$ FSAs, i.e., $\mathcal{A}_1, \ldots \mathcal{A}_{|\mathcal{M}|+3}$.

Product of Finite State Automata. The constraints representing the process behavior, and the conditions on the length of traces must hold true at the same time. This entails that the conjunction of all regular expressions R_1–R_{10} must be verified. In turn, this means that the product of the derived FSAs is the generator of the traces. We will denote this automaton with $\mathcal{A}^{\times} = \mathcal{A}_1 \times \cdot \times \mathcal{A}_{|\mathcal{M}|+3}$.

Generation of Strings. The traces of the event log are created on the basis of the strings accepted by $\mathcal{A}^{\times}$. To this extent, a random path is chosen along $\mathcal{A}^{\times}$ that terminates in an accepting state, and characters of traversed transitions are concatenated. The resulting string corresponds to the backbone of a trace for

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2



(a) *Alt.Prec.*(b, d) (b) *Response*(f, g) (c) *Precedence*(d, a) (d) *Succession*(a, h)

Fig. 3: FSAs accepting the traces that verify Declare constraints.

the event log. The strings are indeed made of characters that uniquely identify activities, in a sequence that complies with the constraints of the input model $\mathcal{M}$.

From Strings to Event Log. We create traces in the event log by deriving the corresponding activities from each character in the strings, keeping the sequences unaltered. Each activity is retrieved through the application of the inverse translation mapping function $\mathscr{I}^{-1}$ to the single-character identifiers. Further information such as timestamps can also be specified for the events: custom attributes can indeed be seamlessly added to enhance the information conveyed by the event log. However, such enrichment goes beyond the scope of this paper.

This procedure is repeated N times, being N the user-specified parameter indicating the number of desired traces in the log. At the end of the N iterations, the log is returned. This last step concludes the overall approach.

4 Evaluation

Our framework has been implemented as a working prototype integrated within the modeling tool Declare designer [17].[1] Fig. 4 shows a screenshot of its main dialog window, where the user can specify the input parameters affecting the length and the number of the traces in the generated log, once the model has been drawn or loaded from an external Declare XML specification file. The output log can be either encoded using XES [1] or MXML (another XML-based standard format for logs), or as plain text.

In order to evaluate the efficiency of the proposed approach, we have run an extensive set of experiments to assess the time to generate event logs of different sizes, following the trailing example provided in Section 2.2. The results are described in Section 4.1. To validate it from the perspective of the effectiveness, we have used as reference models the fracture treatment example process of Fig. 1, and a real case study. We have generated an event log on the basis of each model, and run two different Declare discovery algorithms, in order to check

[1] https://github.com/processmining/synthetic-log-generator

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

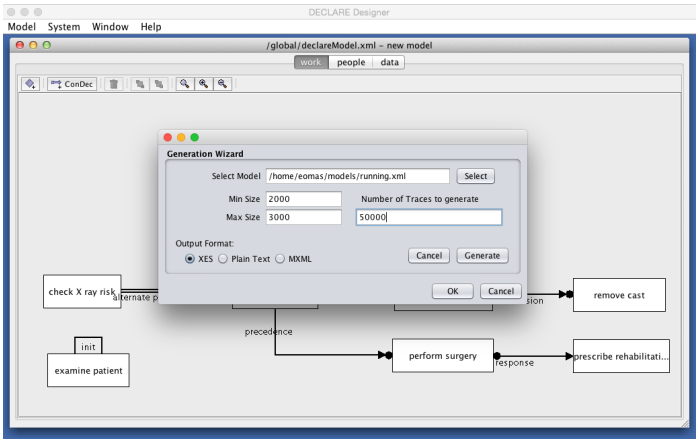


Fig. 4: A screenshot of the implemented prototype.

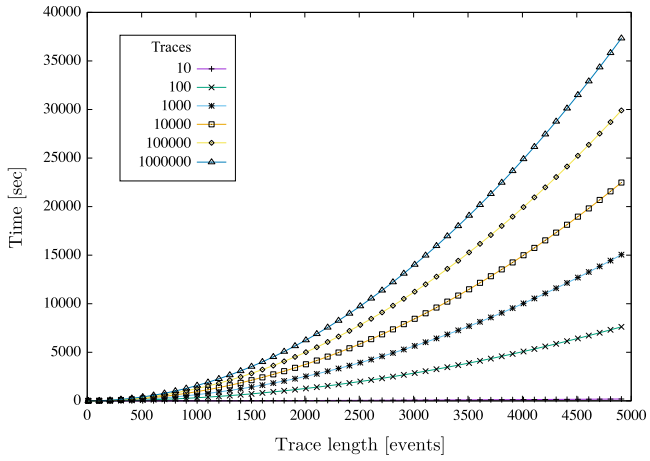


Fig. 5: Generation times with respect to number of traces and trace size.

whether the simulated and the discovered models match. The tests confirm the compliance of the log w.r.t. the input model, as detailed in Section 4.2.

All tests have been conducted on a machine equipped with an Intel i7 CPU quad processor at 2.8Ghz and 12GB of dedicated RAM. We have used Java SE 1.7 as the coding language for the implementation of the framework.

4.1 Efficiency

In order to assess the performance of the log generation approach, we considered as reference model the sample process described in Section 2.2 and depicted in Fig. 1. A set of logs with different characteristics has been generated. In

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

Traces	Trace length			Log size	Trace length		
	500	2500	5000		500	2500	5000
10	2	51	191	10000	242	5875	22477
100	82	1992	7620	100000	323	7816	29906
1000	162	3933	15049	1000000	403	9757	37335

Table 2: The computation time (in seconds) w.r.t. the size of the log.

particular, we have sampled the generation times that resulted by varying the following user parameters: *(i)* the number of events per trace, keeping minimum and maximum values equal ($n_l = m_l$), and *(ii)* the number of traces in the log N. In order to show that the generator can be used in real contexts also to generate very large logs within acceptable times, we increased the number of traces from 10 to 10^6 at a base-10 logarithmic step, and raised the length of each trace from 10 to 5,000 events, at a step of 500 units. The number of total events per log ranged from 100 up to $5 \cdot 10^9$.

In Fig. 5, each curve reports the computation time needed w.r.t. the incremented number of events per trace. Curves are parametric w.r.t. the number of traces per log. The shown trend is a flattened branch of parabola. Table 2 lists the sampled times w.r.t. the logarithmic progression of the number of traces instead, fixing three values for the trace length (500, 2500, and 5,000). As the reader can notice, values increase linearly w.r.t. to the exponentially increasing number of traces in the log, for all three fixed trace lengths. We can thus conclude that the performance increases logarithmically in the number of traces. The moderated ascent of the parabola in Fig. 5 is here confirmed, e.g., by the fact that 242 seconds (approx. 4 minutes) are sufficient to generate a very big log containing $5 \cdot 10^6$ events, distributed along 10,000 traces of 500 events each.

4.2 Effectiveness

To evaluate the effectiveness of the log generation approach, we have carried out two different experiments. In the first experiment, we have used the fracture treatment example process of Fig. 1 to generate a log containing 1,000 traces of length comprised between 2 and 100 events. We exported the log in XES format. Then, we have used the Declare Miner, a ProM plug-in[2] for the discovery of Declare models [14, 15], to mine the log and check whether the discovered Declare model was in line with the simulated one. The discovered model is shown in Fig. 6. The figure shows the list of constraints that are satisfied in 100% of the cases. The model contains the same constraints as the simulated one, thus experimentally confirming the correctness of the generated log.

In our second experiment, we have tried to reproduce with our tool the behavior of a real-life log. To this aim, we have used the log provided for the BPI

[2] www.processmining.org/prom/start

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

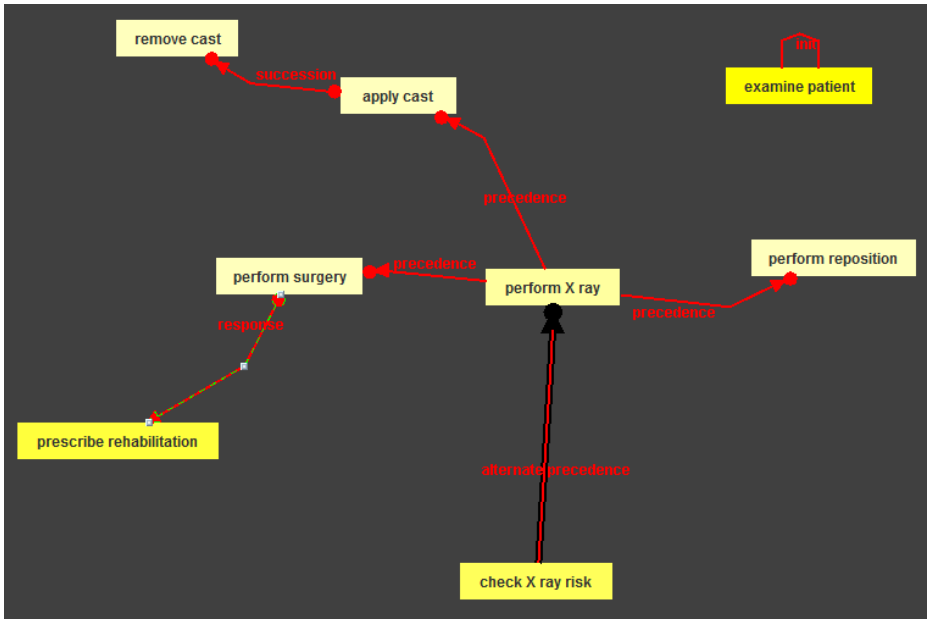


Fig. 6: Discovered model from the log generated starting from the fracture treatment process model.

challenge 2014 by Rabobank Netherlands Group ICT [31]. The log pertains to the management of calls or mails from customers to the Service Desk concerning disruptions of ICT-services. The log contains 46,616 traces and amounts to 466,737 events referring to 39 different activities. We have used the Declare Miner to discover a model from this log. The discovered model is shown in Fig. 7. Starting from this model, we generated a log with 46,616 cases of length between 1 and 173. These parameters were chosen according to the characteristics of the original real life log. We exported the log in XES format. Fig. 8 shows the ProM log visualizer dashboard window, listing the main statistics about the loaded log. In this second experiment, we have used MINERful [32] to rediscover the model. Using again the option that in this tool allows the user to discover only the constraints always satisfied, we obtain the same model shown in Fig. 7.

We can conclude that the logs generated by our approach reproduce exactly the behavior of the input models, regardless of the discovery algorithm adopted to verify it, and irrespectively of whether the aim is to simulate a hand made reference model, or rather to replicate the behavior of an existing real life event log. Furthermore, our tool is highly integrated with the state-of-the-art software for modeling and mining Declare processes.

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

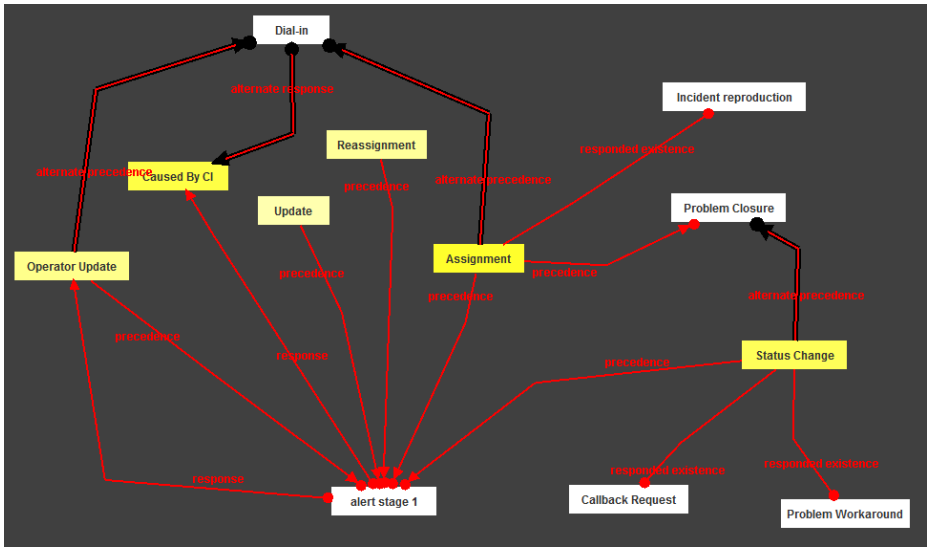


Fig. 7: Discovered model from the log provided for the BPI challenge 2014.

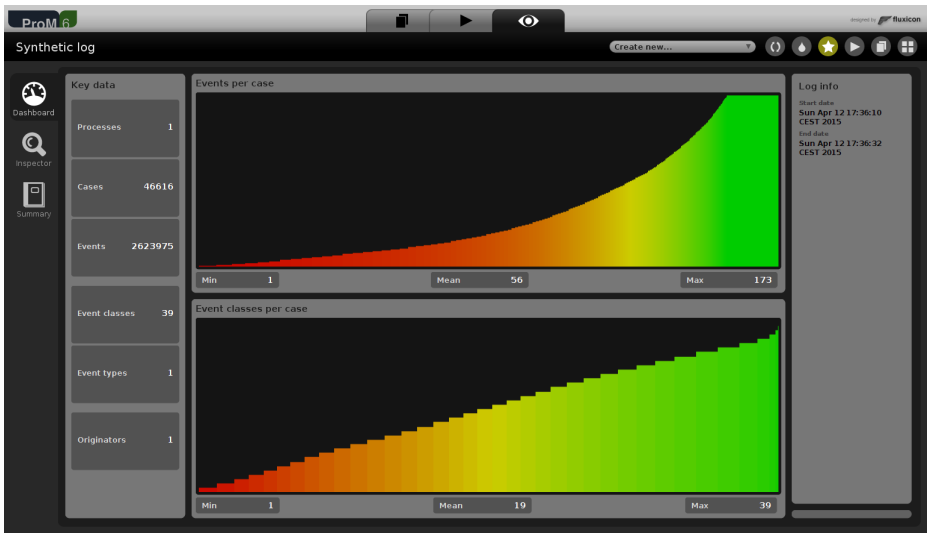


Fig. 8: Information about the log generated starting from the Declare model in Fig. 7.

5 Related work

The automated generation of event logs to test process mining algorithms has been studied extensively in the context of procedural modeling languages. The work of Hee and Liu [8] introduces a framework for the automated generation of

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

classes of Petri nets (PNs), according to user-defined topological rules. Generated Petri nets(PNs) are meant to be used as benchmarks for algorithms. This work is of inspiration for us, in that we also create a graph-based structure as a means to create benchmarking data (event logs). In [33], an approach based on CPN Tools [7] is described, to generate XML event logs by the simulation of a Colored Petri net (CPN). In [6], Burattin and Sperduti propose an approach for the generation of logs. Process descriptions are meant to be provided via a (stochastic) context-free grammar, whose definition is based on well-known process patterns. The work of [6] relates to the one presented in this paper in that we adopt regular expressions to generate logs, and the REs express regular languages, hence, languages accepted by left-linear context-free grammars. However, we use REs as translations of business rules, and not as production rules for the topology of the process.

All the approaches described so far support indeed only procedural business process models. Procedural business process models show some limitations when the represented process is characterized by several possible execution paths, high variability and continuous changes [11], as in the case of knowledge-intensive processes [34]. In this context, declarative process models such as Declare [35] are proven to perform better in terms of compactness [36] and customizability [37]. The newest version of CPN Tools [38] allows the user to graphically add Declare constraints to the transitions of a CPN, thus obtaining hybrid models. The simulation tool allows both user-driven and random executions of such models. Our framework differs from CPN Tools in that it is not an extension of a procedural-based modeler, but inherently a tool for the management of Declare process models, specialized in the generation of event logs. For instance, the number of traces to be generated is here a parameter. In CPN Tools, instead, a workaround would be needed, resorting on the initial marking of user-specified fictitious places linked to process activities/transitions. In addition, notions outside the Declare specification are needed for simulation (marking, places, tokens), and a Declare model could not thus be simply loaded to generate the logs.

Preliminary versions of log generators based on Declare models were presented in [32, 13, 39, 40] for testing the time performance of the proposed process mining tools. A discussion on the adoption of a product of FSAs to represent a conjunction of declarative constraints can also be found in [30]. Here, we present the complete approach, which is detached from any discovery algorithm and can be used for the creation of platform-independent benchmarks.

6 Conclusions and Future Work

In this paper, we have presented an approach for generating event logs based on the simulation of declarative models. The proposed approach is based on the translation of Declare constraints into regular expressions. The framework has been presented in its execution flow, which undergoes a sequence of steps leading from a set of Declare constraints to a simulation automaton, to the final event log. The evaluation has shown that the implemented solution, integrated

Pre-print copy of the manuscript published by Springer
(available at link.springer.com)
identified by DOI: 10.1007/978-3-319-24626-0_2

within the Declare designer tool, correctly reproduces user-defined models and replicates the behavior of existing logs. Two experiments have been conducted as an experimental evidence of such claims, respectively using as input an example reference model, and a model stemming from the BPI Challenge 2014 benchmark event log. The generated logs have been subject to the processing of two different declarative process discovery techniques, and in both cases the retrieved model matched the input one. Performance tests also showed that the algorithm is capable of generating very large logs in a fairly small amount of time. It is in the future plans to extend the framework towards the creation of logs containing user specified data attributes and complex activity life-cycles.

Acknowledgments. The work of Claudio Di Ciccio has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant agreement 318275 (GET Service).

References

1. Verbeek, H., Buijs, J., van Dongen, B., van der Aalst, W.: XES, XESame, and ProM 6. In: Information Systems Evolution. Volume 72. Springer (2011) 60–75
2. Scheer, A., Nüttgens, M.: ARIS architecture and reference models for business process management. In: Business Process Management, Models, Techniques, and Empirical Studies. Volume 1806. (2000) 376–389
3. Scheer, A.: ARIS toolset: A software product is born. Inf. Syst. **19**(8) (1994) 607–624
4. van Dongen, B.: BPI challenge 2011 (2011)
5. van Dongen, B.: BPI challenge 2012 (2012)
6. Burattin, A., Sperduti, A.: PLG: A framework for the generation of business process models and their execution logs. In: BPM Workshops. Springer (2011) 214–219
7. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3) (May 2007) 213–254
8. Hee, K.v., Liu, Z.: Generating benchmarks by random stepwise refinement of petri nets. In Donatelli, S., Kleijn, J., Machado, R., Fernandes, J., eds.: PETRI NETS 2010. CEUR-ws.org (2012) 403–417
9. Bergmann, G., Horváth, A., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: Graph Transformations. Volume 5214 of Lecture Notes in Computer Science. Springer (2008) 396–410
10. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: BPM Workshops. BPM'06, Springer (2006) 169–180
11. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: Enterprise, Business-Process and Information Systems Modeling. Volume 29. Springer (2009) 353–366
12. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: BPM Workshops. Volume 99. Springer (2012) 383–394

Pre-print copy of the manuscript published by Springer
(available at `link.springer.com`)
identified by DOI: `10.1007/978-3-319-24626-0_2`

13. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Trans. Manage. Inf. Syst. **5**(4) (2015) 24:1–24:37
14. Maggi, F.M.: Declarative process mining with the declare component of ProM. In: BPM (Demos). (2013)
15. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: CAiSE. (2012) 270–285
16. Bernardi, M.L., Cimitile, M., Francescomarino, C.D., Maggi, F.M.: Using discriminative rule mining to discover declarative process models with non-atomic activities. In: RuleML 2014. Volume 8620. (2014) 281–295
17. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: full support for loosely-structured processes. In: EDOC. (2007) 287–300
18. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
19. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
20. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers **8**(1) (1998) 21–66
21. van der Aalst, W.M.P.: Verification of workflow nets. In: ICATPN. (1997) 407–426
22. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. **30**(4) (2005) 245–275
23. Di Ciccio, C., Mecella, M., Scannapieco, M., Zardetto, D., Catarci, T.: MailOfMine – analyzing mail messages for mining artful collaborative processes. In: Data-Driven Process Discovery and Analysis. Volume 116. Springer (2012) 55–81
24. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - R&D (2009) 99–113
25. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. (2013)
26. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on ltl on finite traces: Insensitivity to infiniteness. In: AAAI. (2014)
27. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Trans. Comput. Log. **9**(4) (2008) 29:1–29:43
28. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. T. Petri Nets and Other Models of Concurrency **2** (2009) 278–295
29. Gisburg, S., Rose, G.F.: Preservation of languages by transducers. Information and Control **9**(2) (1966) 153 – 176
30. Prescher, J., Di Ciccio, C., Mendling, J.: From declarative processes to imperative models. In: SIMPDA, CEUR-WS.org (2014) 162–173
31. van Dongen, B.: BPI challenge 2014 (2014)
32. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: CIDM, IEEE (2013) 135–142
33. de Medeiros, A.A., Günther, C.W.: Process mining: Using CPN tools to create test logs for mining algorithms. In: Proceedings of the sixth workshop on the practical use of coloured Petri nets and CPN tools (CPN 2005). Volume 576. (2005)
34. Di Ciccio, C., Marrella, A., Russo, A.: Knowledge-intensive Processes: Characteristics, requirements and analysis of contemporary approaches. J. Data Semantics **4**(1) (February 2015) 29–57
35. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM. (2006) 1–23

Pre-print copy of the manuscript published by Springer
(available at `link.springer.com`)
identified by DOI: `10.1007/978-3-319-24626-0_2`

36. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D **23**(2) (2009) 99–113
37. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N., van der Aalst, W.M.P.: Configurable Declare: Designing customisable flexible process models. In: CoopIS. (2012) 20–37
38. Westergaard, M., Slaats, T.: Cpn tools 4: A process modeling tool combining declarative and imperative paradigms. In: BPM (Demos). (2013)
39. Di Ciccio, C., Maggi, F.M., Mendling, J.: Discovering target-branched declare constraints. In: BPM, Springer (2014) 34–50
40. Di Ciccio, C., Mecella, M., Mendling, J.: The effect of noise on mined declarative constraints. In: Data-Driven Process Discovery and Analysis. Volume 203. Springer (2015) 1–24

Pre-print copy of the manuscript published by Springer
(available at `link.springer.com`)
identified by DOI: `10.1007/978-3-319-24626-0_2`

This document is a pre-print copy of the manuscript
(Di Ciccio et al. 2015)
published by Springer
(available at link.springer.com).

The final version of the paper is identified by DOI: 10.1007/978-3-319-24626-0_2

References

Di Ciccio, Claudio, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi (2015). "Generating Event Logs through the Simulation of Declare Models". In: *EOMAS*. Ed. by Joseph Barjis, Robert Pergl, and Eduard Babkin. Vol. 231. Lecture Notes in Business Information Processing. Springer, pp. 20–36. ISBN: 978-3-319-24625-3. DOI: 10.1007/978-3-319-24626-0_2.

BibTeX

```
@InProceedings{   DiCiccio.etal/EOMAS2015:GeneratingEventLogs,
  author        = {Di Ciccio, Claudio and Bernardi, Mario Luca and Cimitile,
                   Marta and Maggi, Fabrizio Maria},
  title         = {Generating Event Logs through the Simulation of {D}eclare
                   Models},
  booktitle     = {EOMAS},
  year          = {2015},
  pages         = {20--36},
  publisher     = {Springer},
  crossref      = {EOMAS2015},
  doi           = {10.1007/978-3-319-24626-0_2},
  keywords      = {Declare; Regular Expressions; Declarative Process Models;
                   Process Simulation; Log Generation}
}
@Proceedings{     EOMAS2015,
  title         = {Enterprise and Organizational Modeling and Simulation -
                   11th International Workshop, {EOMAS} 2015, Held at CAiSE
                   2015, Stockholm, Sweden, June 8-9, 2015, Selected Papers},
  year          = {2015},
  editor        = {Joseph Barjis and Robert Pergl and Eduard Babkin},
  volume        = {231},
  series        = {Lecture Notes in Business Information Processing},
  publisher     = {Springer},
  isbn          = {978-3-319-24625-3},
  doi           = {10.1007/978-3-319-24626-0}
}
```