UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Vasyl Skydanienko

Data-aware Synthetic Log Generation for Declarative Process Models

Master's Thesis (30 ECTS)

Supervisor(s):
Fabrizio Maria Maggi
Chiara Di Francescomarino
Chiara Ghidini

Tartu 2018

Data-aware Synthetic Log Generation for Declarative Process Models

Abstract:

In Business Process Management, process mining is a class of techniques for learning process structure from an execution log. This structure is represented as a process model: either procedural or declarative. Examples of declarative languages are Declare, DPIL and DCR Graphs. In order to test and improve process mining algorithms a lot of logs with different parameters are required, and it is not always possible to get enough real logs. And this is where artificial logs are useful. There exist techniques for log generation from DPIL and declare-based models. But there are no tools for generating logs from MP-Declare – multi-perspective version of Declare with data support. This thesis introduces an approach to log generation from MP-Declare models using two different model checkers: Alloy and NuSMV. In order to improve performance, we applied optimization to baseline approaches available in the literature. All of the discussed techniques are implemented and tested using existing conformance checking tools and our tests. To evaluate performance of our generators and compare them with existing ones, we measured time required for generating log and how it changes with different parameters and models. We also designed several metrics for computing log variability, and applied them to reviewed generators.

Keywords:

Business process, simulation, process model translation, declarative model, Declare, Alloy, first order logic, log generation

CERCS: T120 Systems engineering, computer technology

Andme toega sünteetilise logi genereerimine deklaratiivsetele protsessimudelitele

Lühikokkuvõte:

Äriprotsesside juhtimises on protsessikaeve klass meetodeid, mida kasutatakse protsessi struktuuri õppimiseks täitmislogist. Selle struktuur on esindatud kui protsessi mudel: kas menetluslik või deklaratiivne. Näited deklaratiivsetest keeltest on Declare, DPIL ja DCR Graphs. Selleks, et testida ja parandada protsessi kaevandamise algoritme on vaja palju logisid erinevate parameetritega ja alati ei ole võimalik saada piisavalt reaalseid logisid. See on koht, kus tehislikud logid tulevad kasuks. On olemas meetodeid logi genereerimiseks DPIL-ist ja deklaratiivsetest mudelitest, kuid puuduvad vahendid logi genereerimiseks MP-Declare-ist, mis on multiperspektiivne versioon Declare-ist andmete toega. Käesolev magistritöö käsitleb MP-Declare mudelitest logide genereerimist kasutades kaht erinevat mudelite kontrollijat: Alloy ja NuSMV. Selleks, et parandada jõudlust, optimeerisime kirjanduses saadaval olevaid baaslähenemisi. Kõik käsitletud tehnikad implementeeritakse ja testitakse kasutades saadaval olevat sobivuse testimise tööriistu ja meie enda väljatöötatud teste.
Meie generaatorite hindamiseks ja võrdluseks olemasolevate lahendustega mõõtsime me logide genereerimise aega ja seda, kuidas see muutub erinevate parameetrite ja mudelitega. Me töötasime välja erinevad mõõdupuud logide varieeruvuse arvutamiseks ja rakendasime neid uuritavatele generaatoritele.

Võtmesõnad:

Äri protsessi, simulatsiooni, protsessi mudel tõlge, deklaratiivne mudel, Declare, Alloy, esimest järku loogika, logi põlvkonna

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Table of Contents

1 Introduction

In Business Processes Management, process mining stands for a set of techniques designed to analyze and enhance processes by using data from previous executions. This data is stored in event logs by information systems supporting the processes. In process mining we can apply mining algorithms to learn the structure of the process from the log. Using this information about the process behavior, we can identify the problems of our processes and make them more efficient. Learning a model from a log is a branch of process mining called process discovery, and usually it is the first step in analyzing processes. When model is known, it can be used for conformance checking/process monitoring and performance mining. During conformance checking we can verify that the model is following the process, and find deviations. Process monitoring is similar to conformance checking, but applied to running processes. It detects a violation immediately after or even before it happens. Performance mining allows us to use time information from the process (duration of/between activities, cycle time, waiting time, etc.) to identify the bottlenecks and use this information for process redesign.

In order to develop better process mining techniques it should be possible to compare and evaluate them. Usually, mining algorithm under testing tries to mine different logs generated using a known model, and then the discovered behaviors compared with the original model. Though there are some logs publicly available, most companies don't share them, e.g., due to privacy issues. Also, real logs are not always good for testing, because we do not have control over their different parameters like length of traces, amount of noise, etc. To make testing more complete and predictable, artificial logs can be used.

There are two major types of languages for modelling processes: procedural and declarative. Imperative business process modelling languages like BPMN are good for describing routine processes with low variability of execution. They consist in a graph containing all possible execution paths. With more flexible processes it is hard to use procedural languages for modelling, because the graphs become too large and hard to understand. Declarative languages allow to do it much easier. A declarative language consists of constraints, and any behavior is allowed, unless it violates at least one constraint.

Most of the modelling languages and tools for process mining use only the control flow perspective of a process. In real life, activities may contain data, like who was executing the activity, what type of resources was used or what amount was paid (e.g. in a selling process), etc. This makes testing mining algorithms using real-life logs even harder, because the amount and type of data cannot be changed, and hence different aspects of the algorithm cannot be tested separately.

Currently there are very few tools available for generating artificial logs from declarative process models. Though some of them support resources, none of them allow to use arbitrary constrained data and no tools support the generation of logs from the full set of MP-Declare (Multi Perspective Declare) constraints. Therefore, we investigate and evaluate the existing implementations, and use this information for designing a better log generator with complete Declare support and data.

The generation of logs from declarative models is not an easy task, and having data support means that we cannot use some known algorithms, e.g., FSA-based algorithms. There are no well-defined execution paths like in imperative models, different constraints might influence and contradict each other, and some models do not even have valid finite traces. It might be hard to find traces for some models due to the combinatorics of the problem, so it is important for the log generator to work efficiently.

When generating a log, it is preferable to get diverse traces rather than similar ones, because this way the log contains more useful information about possible process execution patterns, and the chances that all constraints are fairly activated throughout the log are increased.

In order to address these problems four research questions are investigated in this thesis:

1. How can we encode declarative models with data?
2. How can we efficiently generate event logs from MP-Declare models?
3. How do different approaches and techniques perform in the generation of event logs from data-aware Declare models in terms of generation time and log variability?
4. How can we measure quality of generated logs?

In this thesis, we propose two approaches to log generation from MP-Declare models using Alloy [20] and NuSMV model checkers, and we compare our approaches with state of the art techniques solving the same problem.

As it comes from the name, model checkers allow to check models – to verify that a given model meets certain specifications. In our case, the model is correct when at least one trace compliant with the model exists, which is not always the case as declare constraints may contradict each other (e.g., existence and absence of the same activity). Some model checkers go beyond this, and provide an example or a counterexample for a given model, which proves that it is valid. In our solution we use these examples/counterexamples to generate traces and save them in a suitable log format. Also we introduce a new format for specifying MP-Declare models, and use this format to provide the input for our log generator.

To evaluate the performance of the generators, we came out with a wide experimentation, and developed a set of metrics for measuring the log variability.

In the evaluation, we compared our solutions with each other, and with existing state of the art approaches. In details, we looked at available features (available generation parameters, supported constraints), readiness (is it complete product, prototype, demo, or not implemented), generation time and log variability.

The structure of this paper is the following: Section 2 describes background information needed to understand main concepts used in this paper. Section 3 shows our proposed approaches for the generation of artificial logs from MP-Declare models. Section 4 gives a short overview of the existing literature on this topic. Section 5 and 6 describe the details of the implementation using Alloy and NuSMV, and how they were used in our generators. Section 7 illustrates the tool – interfaces of generators and modeler. Section 8 presents the metrics for the evaluation of log variability, and the comparison results for logs generated by different generators. Section 9 contains the evaluation and the comparison of our and other existing generators in terms of execution times. Section 10 concludes the paper and outlines directions for future work.

2 Background

This section presents some basic definitions, which will be needed for understanding the rest of this paper.

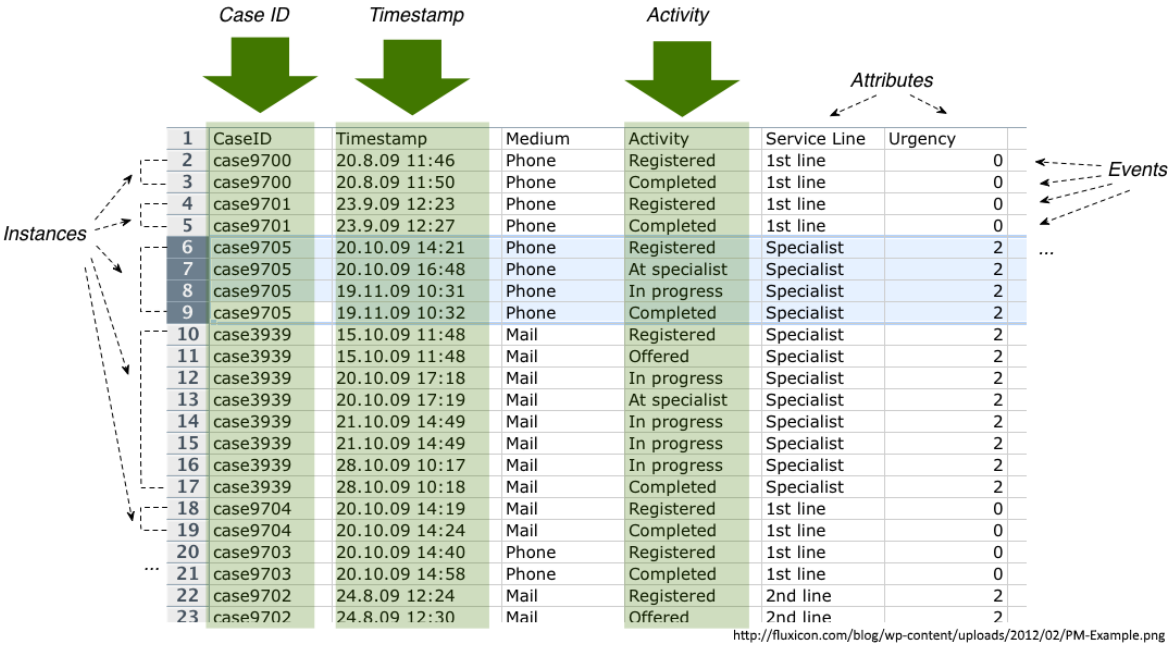
2.1 Process mining

Process mining is a family of techniques aimed at analysing business processes from execution logs.

Log

A process log is a set of traces. Each trace describes one execution of a process as a sequence of events.

The Figure 1 shows an example of an event log as a table. Each line represents one event. Events with the same case id form a trace. Here all the events grouped by case id and sorted by timestamp, so we can clearly see separate traces.



	CaseID	Timestamp	Medium	Activity	Service Line	Urgency
2	case9700	20.8.09 11:46	Phone	Registered	1st line	0
3	case9700	20.8.09 11:50	Phone	Completed	1st line	0
4	case9701	23.9.09 12:23	Phone	Registered	1st line	0
5	case9701	23.9.09 12:27	Phone	Completed	1st line	0
6	case9705	20.10.09 14:21	Phone	Registered	Specialist	2
7	case9705	20.10.09 16:48	Phone	At specialist	Specialist	2
8	case9705	19.11.09 10:31	Phone	In progress	Specialist	2
9	case9705	19.11.09 10:32	Phone	Completed	Specialist	2
10	case3939	15.10.09 11:48	Mail	Registered	Specialist	2
11	case3939	15.10.09 11:48	Mail	Offered	Specialist	2
12	case3939	20.10.09 17:18	Mail	In progress	Specialist	2
13	case3939	20.10.09 17:19	Mail	At specialist	Specialist	2
14	case3939	21.10.09 14:49	Mail	In progress	Specialist	2
15	case3939	21.10.09 14:49	Mail	In progress	Specialist	2
16	case3939	28.10.09 10:17	Mail	In progress	Specialist	2
17	case3939	28.10.09 10:18	Mail	Completed	Specialist	2
18	case9704	20.10.09 14:19	Mail	Registered	1st line	0
19	case9704	20.10.09 14:24	Mail	Completed	1st line	0
20	case9703	20.10.09 14:40	Phone	Registered	1st line	0
21	case9703	20.10.09 14:58	Phone	Completed	1st line	0
22	case9702	24.8.09 12:24	Mail	Registered	2nd line	2
23	case9702	24.8.09 12:30	Mail	Offered	2nd line	2

Figure 1 – Event log in csv format

A typical process log contains a case id, a timestamp and an activity for each event. In the example in Figure 1 we see additional columns with data attributes: medium, service line and urgency. Different processes can have different data attributes. If the value of an attribute never changes within one trace (like in our example), we can say that it is a trace attribute. Otherwise it is an event attribute.

A process log can be stored as a XES [18] (Extensible Event Stream) file, which contains information about the process executions. An extract of a XES file is the following:

```
…
<trace>
        <string key="concept:name" value="Synthetic trace no. 000"/>
        <event>
                <string key="concept:name" value="rehabilitation"/>
```

```
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2016-04-26T16:28:06.903+03:00"/>
            <string key="duration" value="8"/>
        </event>
        <event>
            <string key="concept:name" value="x-ray"/>
```
…

The root node is <log>. Inside the log there is a list of traces in <trace> tags. Each trace may have trace attributes and contains a list of events in the <event> tag. Events typically have a name and a timestamp. The case id is stored in the “concept:name” trace attribute. “duration” is an event attribute, with value equals to 8. The logs generated with our tool are stored in XES files.

Declarative process models

There are two ways currently used for specifying a process model: imperative and declarative.

In an imperative process model, we specify all possible execution paths. If some behavior is not specified in the model, then it cannot happen. Examples of imperative process modelling languages are BPMN and Petri net.

In declarative process model, we have the opposite approach: anything is allowed if it does not violate any rule (constraint) of the model. Example modelling languages are Declare, DPIL and DCR Graphs.

A trace is considered to be compliant with a declarative model iff. it does not violate any rules in the model.

Declare is a declarative process modelling language. It has two representations: graphical and textual. In this paper, the graphical representation will not be used.

To understand how Declare works, consider two activities: A and B (e.g. Apply to University and Become graduated).

If we do not have any constraints, then occurrence of the tasks in any order will produce a valid trace, for example:

1. AAA
2. ABABABBA
3. BBAA
4. BB

But if we add the Precedence(B,A) constraint (which means, that before the execution of B, A should be executed), then the 3-rd and 4-th trace will no longer be compliant.

Declare has a number of constraint templates, which restrict the possible control flow of a process. Table 1 shows the Declare constraints and their description.

Init(A)	First task is A
Existence(A)	Task A should be executed

Existence(A,N)	Task A should be executed N or more times (N is number)
Absence(A)	Task A should not be executed
Absence(A,N)	Task A may be executed N times or less
Exactly(A,N)	Task A should be executed (exactly) N times
Choice(A,B)	Task A or task B should be executed (or both)
ExclusiveChoice(A,B)	Task A or task B should be executed, but not both
RespondedExistence(A,B)	If task A executed, task B executed as well
Response(A,B)	If task A executed, task B executed after A
AlternateResponse(A,B)	If task A executed, task B executed after A, without other A in between
ChainResponse(A,B)	If task A executed, task B executed next
Precedence(A,B)	If task A executed, task B was executed before A
AlternatePrecedence(A,B)	If task A executed, task B was executed before A, without other A in between
ChainPrecedence(A,B)	If task A executed, previous executed task was B
NotRespondedExistence(A,B)	If task A executed, task B is not executed
NotResponse(A,B)	If task A executed, task B will not be executed after A
NotPrecedence(A,B)	If task A executed, task B was not executed before A
NotChainResponse(A,B)	If task A executed, task B is not executed next
NotChainPrecedence(A,B)	If task A executed, previous executed task was not B

Table 1 – Declare constraints

Some models may not have any compliant traces of finite length (they are overconstrained).

Suppose that we have tasks A B and C, and constraints ChainResponse(A,B) and NotResponse(A,B). This model cannot have task A in it, because it will lead to a conflict between the 2 constraints. If we add Existence(A) to it, then there will be no possible traces.

Another example is: Existence(A), Response(A,B), Response(B,A). In this model there are no possible finite traces, as A and B need to interleave infinitely.

In the multi-perspective version of Declare (MP-Declare) tasks may include data, and constraints may constrain this data. For example, if we have a task ‘buy ticket’ with attached the price (data of type float), then a possible constraint which includes data can be Existence(buy ticket) [price<10.0]

All declare constraints can be expressed as LTL. LTL is linear temporal logic model. It allows to use five temporal operators: next (X), globally (G), finally (F), until (U) and release (R). X, G and F are unary operators, U and R are binary. Their meaning is following:

X x – x should be true in the next state

G x – x should be true in all states

F x – x should be true in current or at least one of the next states

x U y – x should be true at least until y becomes true. y should become true.

x R y – y should be true until and at the state x becomes true. x may not become true

2.2 Model checkers

Model checkers are tools for checking the model correctness. In our case a model is an MP-Declare process model. We will use two model checkers: Alloy analyzer and NuSMV. They can produce counterexamples which demonstrates violation of the model constraints (Alloy can also produce valid examples). By using them we can build traces compliant with the original model.

Alloy

Alloy allows to describe a logical structure, and to find a model(s), which fits the described structure. Or it can find a counterexample – a model which does not fit the described structure.

If we describe the notion of activities, trace, MP-Declare constraints in Alloy, it will find compliant traces which can be then transformed into a log.

Internally, Alloy translates an input model to a boolean formula, and then uses SAT solver to get a solution. Its advantage is that it can find exhaustively all possible models which are compliant with the given formula.

A simple example of an Alloy model for a trace with two events with possible activities A and B, and constraint Response(A,B) is the following:

```
abstract sig Activity {}     // this is definition of activity
abstract sig Event{          // this is one event in trace
       name: one Activity    // each event is related to activity
}

one sig A extends Activity {}// activity A
one sig B extends Activity {}// activity B

one sig T1 extends Event {}  // first event of the trace
one sig T2 extends Event {}  // second event of the trace

fact {not T2.name=A and (T1.name=A implies T2.name=B)}    // Response(A,B)
// second event is not A (because in this case it will not be followed by B),
// and if first is A then second is B
```

If we run this example in the Alloy analyzer, we will get two results: AB and BB.

Note, that this example does not define any order of events in the trace, we just implicitly imply that T2 comes after T1. Also it does not include any data. Nonetheless, this example is close to the actual implementation which will be described in section 4.

NuSMV/NuXMV and LTL

NuSMV and NuXMV are symbolic model checkers. They allow to encode a model and then check it with LTL constraints. If the model violates the constraints, the tool produces a counterexample trace. The main difference between NuXMV and NuSMV is that NuSMV uses only a SAT solver (similarly to Alloy), while NuXMV can also use an SMT solver, which allows the use of infinite domain variables. As it can only produce counterexample, in our generator we need to negate all the constraint, so counterexample of negated model will be example of the model.

3 Problem and Approach

Recently, more process logs containing data are becoming available. In order to work with these logs, different tools started to appear. One category of such tools is the one of synthetic log generators.

Log generators are needed for different sorts of tasks. Some of them are: testing process mining algorithms [1][5][22], translation between process model languages (M2MT) [2][3], declarative process visualization [6]. When testing mining algorithms the standard procedure is the following: a log is generated starting from a model, then another model is mined from this log and the initial model should match the mined one. One of the ways to do model-to-model translation is to generate a log from one model, and then mine this model in another notation. Model visualization allows to see potential scenarios of process executions, to better understand their behavior [21]. In some cases, it is more suitable to use synthetic logs, because real logs may contain noise, deviations and other imperfections.

Though log generation tools for declarative process models are needed in the above scenarios, data support in the currently existing tools is not present or it does not cover the entire semantics of MP-Declare.

In this thesis, we want to answer the following research questions were formulated:

1. How can we encode declarative models with data?
2. How can we efficiently generate event logs from MP-Declare models?
3. How do different approaches and techniques perform in the generation of event logs from data-aware Declare models in terms of generation time and log variability?
4. How can we measure quality of generated logs?

To answer these research questions we propose a new format for expressing MP-Declare models (RQ1), two log generators based on two model checkers (Alloy and NuSMV) (RQ2, RQ3), and a set of metrics to evaluate the quality of the logs (RQ4).

The log generator based on Alloy is implemented following the approach [16] shown in Figure 2:

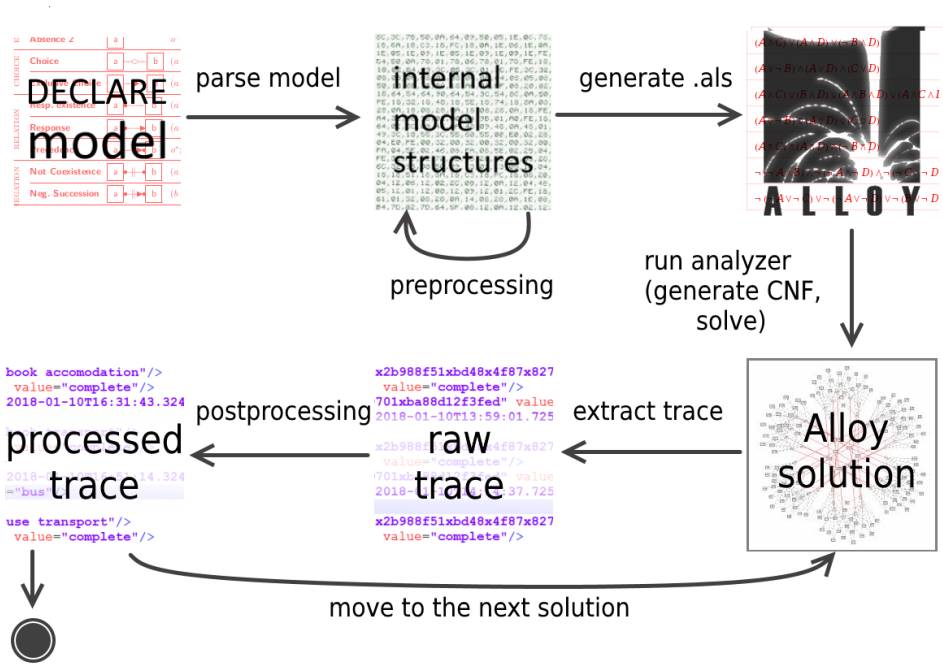


Figure 2 – Alloy based generator architecture

In the first step we parse the MP-Declare model – deriving separate statements in groups (activity definitions, constraints, etc.), build expression trees for functions of data constraints, define mapping between activities and data.

In the preprocessing step we map names to identifiers (to allow the usage of names with characters not supported by Alloy) and generate intervals for numeric values according to constraints since Alloy does not support numeric variables (explained in more details in section 4).

After the preprocessing we generate the Alloy code corresponding to the MP-Declare model. Then we run the Alloy analyzer. The analyzer generates the CNF (conjunctive normal form) for the model, and then solve it using a SAT (boolean satisfiability problem) solver [17].

In Alloy, when a model has several constraints on the same activity, the one which comes first activated more frequently. This causes generation of unbalanced logs. To overcome the problem, we introduced an option for the constraint shuffling. With this option enabled generator generates the log in several iterations, shuffling constraints order in between.

When we have the Alloy solution, we can explore it and extract the trace. Finally, in the post-processing stage, we remove empty activities, replace intervals with actual numbers and decode all names.

We then move to the next solution and repeat the last two steps until the required amount of traces is collected. When all traces are generated, they are saved in a file.

The log generator based on NuSMV works differently as shown in Figure 3

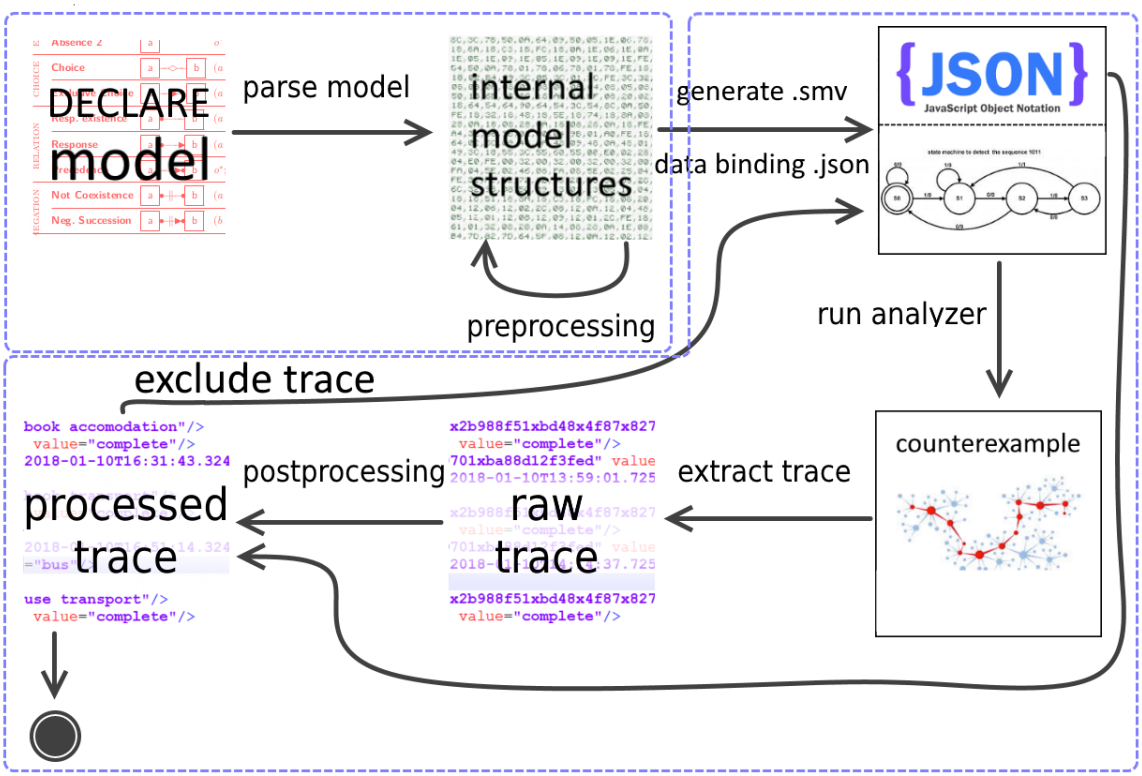


Fig 3 – SMV-based generator architecture

The first two steps are the same as in the Alloy-based solution. The SMV generator uses the same format of MP-Declare model.

In the code generation step we create two files. First, a .smv model for the NuSMV tool. Second, a .json file, which describes data binding to activities and is used internally. Indeed, due to the peculiarities of NuSMV, this information is not used by the model checker.

In the next step we run the model checker, which produces a counterexample for the generated model (where all the constraints are negated), so the counterexample is actually an example for input model.

When a counterexample is obtained, it is possible to extract a trace from it and save it in XES format.

After the generation of one trace, second run of the generator will produce exactly the same trace. To generate different traces, we modify a model in a way that it disallows the last trace.

4 Related work

In this section we will discuss existing papers on the topic. We will take a closer look on the described solutions, their applicability to our problem, and find some baselines to improve.

There are numerous approaches to the simulation and log generation for business processes. In [8] the authors describe the concept of abductive logic reasoning, and use SCIFF framework for log generation. Both: declarative and imperative process models can be used, but no translation from process model to SCIFF has been implemented. Overall [8] is in very early stage and cannot be evaluated or used yet.

The works in [9], [10] and [11] are three consecutive works about log generation from BPMN models (imperative). The approach of the authors is to simulate the process by maintaining a set of enabled activities, and iteratively putting a randomly chosen activity from the set into a trace. After each iteration the set is updated. Additionally, this type of simulation allows to specify the duration of activities, the time between activities and probabilities for control flow splits.

There are also generators based on mixed model. In [12] authors show CPN tool, which allows adding declarative constraints to the transitions of colored Petri net.

The works discussed in previous paragraphs of this section are mostly targeting procedural models. The following approaches are intended for declarative models, and use similar approaches to ours.

In [1,2] the authors present a log generation technique from multi-perspective process models based on DPIL (declarative process intermediate language).

Table 2 shows the list of possible constraints in language:

Macro	**Expanded Pattern**	**Semantic**
sequence(a,b)	event(of b at $:t$) implies event(of a at $< t$)	Task b cannot be started before task a has been completed.
once(a)	event(of a at $:p$) implies not event(of a at $< p$)	Task a can be started once only.
consumes(c,i)	event(of c at :t) implies write(of i at $< t$)	Task c can not be started before a data object i has a value.
produces(p,o)	event(of p :t) implies write(of o at $< t$)	Task p can not be completed before a value for the data object o is present.
role(a,r)	event(of a by $:id$) implies relation(subject id predicate $hasRole$ object r)	Task a must be performed by a process participant in role r.
binding(a,b)	event(of b by $:id$) implies event(of a by id)	The tasks a and b must be processed by the same identity.

Table 2 – DPIL constraints

We can see that this language supports some data and has some constraints similar to Declare ones. Sequence(a,b) is equivalent to precedence(a,b), once(a) is equivalent to exactly(1, a). Consumes, produces, role and binding are data constraints, but the restriction here is that data types can only be a document and a role. Therefore, there are two types of data constraints.

After describing the model using DPIL, the first step toward the log generation is to translate it into another language – Alloy. Alloy is a declarative language for describing models. After the model is described we can do two things: try to find a valid example which satisfies our

assertion and try to find counterexample which proves that our assertion is wrong. For the log generation we need the first one – find instances which satisfy our model.

In order to convert a declarative process model into Alloy we need first to define what a trace is, a task is, etc. in Alloy terms. After this we can convert the rules themselves. The paper contains the definition of the basic structures required, and also conversion rules for the constraints presented before.

In [2] the authors present techniques for translating models from DPIL to BPMN and backwards.

For translating models, they employ the following two steps:

1. Generate traces from the model.
2. Mine models in a different format from traces. For trace generation in this paper authors use the technique described in [1].

Similarly to [2], in [3] authors also describes a model translation between DPIL and BPMN using simulation. But they use a different simulation technique.

First they map all the task in a model to characters. Then they convert each constraint into a regular expression. For each regular expression they get corresponding finite state automata (FSA). After this they compute the product of all FSAs. Finally, when they have the product, they can traverse it through with random path, and each path will correspond to one trace. The work in [5] is focused on the generation part in more details using this technique. The disadvantage is that we cannot add data to it.

5 A new format for MP-Declare

To provide a process model as an input to the generator, we need to have some file format for storing it. As existing XML-based format for graphical tools for editing Declare models like the "Declare designer" do not support data, we designed a format for defining the model which can be used for storing it as a plain text file. The format is a structured textual format since extending the graphical representation of Declare would make the model difficult to understand.

There are 6 types of statements which can be written in the model: the definition of an activity, the definition of a data attribute, the binding of an activity to data, a constraint, a data constraint and the definition of a trace attribute.

Here is the specification of the language:

Definition of a Declare activity:

```
activity activity_name
activity_name :: name
```

Example:

```
activity SubmitApplication
```

There are three types of data which can be defined: enumerative, integer and float.

Definition of enumerative data:

```
name: values
values ::
        data_value |
        values, data_value
data_value :: name
```

Example:

```
TransportType: Car, Plane, Train, Bus
```

This will create data with key (attribute name in a log) "TransportType". Value can be either Car, Plane, Train or Bus.

Definition of numeric data:

```
num_data_name: num_type between number and number
num_data_name :: name
num_type :: one of
        integer float

number ::
        digit |
        number digit
```

(the definition of number is simplified for brevity; a float attribute can have values with decimals)

Example:

```
Price: integer between 0 and 300
Angle: float between 0 and 180
```

This will create two data attributes with keys “Angle” and “Price”

Binding data to activity (iff activity has binded data, all ocurrences of this activity in a trace will have this data):

```
bind activity_name: data_name
```

Example:

```
bind AssessApplication: AssessmentType, AssessmentCost
```

Declare constraint:

```
constraint ::
        unary_constraint[activity_name] |
        unary_constraint[activity_name]|function |
        unary_nconstraint[activity_name, number] |
        unary_nconstraint[activity_name, number]|function |
        binary_constraint[activity_name] |
        binary_constraint[activity_name]|function|function


unary_constraint :: one of
        Init Existence Absence


unary_nconstraint :: one of
        Existence Absence Exactly


binary_constraint :: one of
        Choice ExclusiveChoice RespondedExistence Response AlternateResponse ChainRe-
sponse Precedence AlternatePrecedence ChainPrecedence NotRespondedExistence NotResponse
NotPrecedence  NotChainResponse NotChainPrecedence


function ::
        empty |
        function or function |
        function and function |
        not function |
        ( function ) |
```

```
        same data_name |
        different data_name |
        variable . data_name is data_value |
        variable . data_name is not data_value |
        variable . data_name in ( values ) |
        variable . data_name not in ( values )
        variable . num_data_name comparator number

comparator :: one of
        > < >= <= =

variable :: one of
        A B
```

A corresponds to the first activity of the Declare constraint, and B to the second one (see examples).

The variable name (A and B) can be overwritten for data constraints by adding a new name after the activity name. E.g., for unary constraint it will look like this:

```
unary_constraint[activity_name variable_name]|function
```

Example:

```
        Exactly[bookTransport T, 2]|T.type is car
```

Other examples of the format:

```
        Exactly[bookTransport, 2]
```

This is a Declare constraint without data. Activity bookTransport should occur exactly twice. If it has data attached, the value can be any.

```
        Exactly[bookTransport, 2]| A.price<100 and A.transportType in (car, plane)
```

This is an MP-Declare constraint. Activity bookTransport should occur exactly two times with value of price less than 100, and with value of transportType either equal to car or plane. More occurrences of bookTransport are possible, but with different data values (price>=100 or transportType not in (car, plane))

```
        ChainResponse[bookTransport, useTransport] || same transportType
```

This is an MP-Declare constraint. Activity bookTransport should be immediately followed by activity useTransport. Data items with key transportType should have the same values in both these activities.

```
        ChainResponse[bookTransport, bookTransport] | A.transportType is Plane |
B.transportType is Car
```

This is an MP-Declare constraint. Activity bookTransport where transportType has value Plane should be immediately followed by activity bookTransport with transportType equal to Car.

6 The Alloy-based log generator

6.1 A baseline solution investigation

In order to present our encoding of the MP-Declare rules in Alloy we will start investigating the existing solutions and, in particular, the solution presented in [1]. In this subsection we discuss what has been implemented, what problems the solution in [1] has, and then, in the next subsection, we extend and improve it.

Model. In [1] the authors use an example of a "business trip" process in DPIL. Normal flow of this process is following: somebody applies for the trip, gets an approval for it, requests accommodation and transport and collect tickets. After trip is done related documents are archived. In this paper we will give the Declare version of their model. The model is:

```
Precedence(Approve application, Book means of transport)
Precedence(Approve application, Book accommodation)
Precedence(Book means of transport, Collect tickets)
Precedence(Book accommodation, Collect tickets)
Absence(Apply for Trip, 1)
Absence(Approve application, 1)
Absence(Collect tickets, 1)
Absence(Archive documents, 1)
Precedence[Approve application, Apply for Trip B]||B.Application is Application.written
Precedence[Archive documents, Apply for Trip B]||B.Application is Application.written
Precedence[Archive documents, Collect tickets B]||B.TicketCollection is TicketCollec-
tion.written
NotRespondedExistence[Book means of transport, Apply for Trip]||different org::resource
NotRespondedExistence[Book accommodation, Apply for Trip]||different org::resource
NotRespondedExistence[Collect tickets, Apply for Trip]||different org::resource
NotRespondedExistence[Archive documents, Apply for Trip]||different org::resource
```

The typical execution trace for this model will be following:

```
(Apply for Trip, Approve application, Book means of transport, Book accommodation, Collect
tickets, Archive documents)
```

Now let's discuss how did the log generator for this model was implemented in [1].

Similarly to classes in object-oriented languages, in Alloy we have signatures. These signatures can be abstract.

For example, in [1] tasks of the model were encoded in the following way:

```
one sig ApplyForTrip extends Task {}
one sig ApproveApplication extends Task {}
one sig BookMeansOfTransport extends Task {}
one sig BookAccomodation extends Task {}
one sig CollectTickets extends Task {}
one sig ArchiveDocuments extends Task {}
```

We see, that each task extends the Task signature. In the Alloy the parent signature has somewhat different meaning from superclasses in OOP. The parent signature is the set of

all its children. So here (Task) = (Apply for Trip + Approve application + .. + Archive documents), where + is the union operation.

The 'one' keyword before sig means singleton -- only one instance of this entity can exist in the model (though it can be referenced from many places). Other possible options are 'no' (no instances), 'lone' (0 or 1 instance), 'some' (one or more), or undefined. In case of 'some' or undefined multiplicity, we will need to specify the maximum amount of instances before starting simulation.

The definition of Task is following:

```
abstract sig Task extends AssociatedElement{}
```

And AssociatedElement:

```
abstract sig AssociatedElement { }
```

Note that we are omitting some parts here, which are related to roles and resources (you can find them in the original paper). For now, AssociatedElement = Task.

Trace. What we have seen so far is just a set of tasks, without any order or constraint. Now we give the trace definition.

```
abstract sig PEvent {
	pos: disj Int
}
one sig StartEvent extends PEvent{}
one sig EndEvent extends PEvent {}
sig TaskEvent extends PEvent{
	assoEl: some AssociatedElement
}{
	#(Task & assoEl) = 1
}
```

(I made TaskEvent non-abstract unlike original definition, to make it more comprehensible)

What we have here ise a trace -- one StartEvent, one EndEvent, and some events in between. Each event has a position in thtrace (pos: disj Int). 'Disj' means that the position is unique (no two events are in the same position). In the signature definition, the first curly brackets define the block of variables in the signature, and the second ones define the constraints applicable to this signature.

Each TaskEvent (event between start and end) is linked to one task. '#(Task & assoEl) = 1' means that exactly one Task is in the set of associated elements of TaskEvent (field assoEl; # is a cardinality operator – returns count of items in a set. & is intersection of the sets. In the original code there are other types of associated elements which we omitted).

Finally, there are three constraints aiming to ensure that all the events have a sequential position between start and end:

```
// StartEvent has the lowest possible integer
fact { all intVal: Int | intVal >= StartEvent.pos }


// All events other than the StartEvent/EndEvent have a position greater than 0 and smaller
than EndEvent.pos (sequence)
fact{
```

```
        all e: (PEvent - StartEvent - EndEvent) | e.pos < (StartEvent.pos + #TaskEvent + 1)
}


// EndEvent is the last PEvent; finally leads to: Position increment is 1
fact{
        EndEvent.pos <= (StartEvent.pos + #TaskEvent + 1)
}
```

The notes to the code above are given in the comments which start with '//'. The only thing to add is that the '-' operator in Alloy means difference of sets, and '|' can be read as 'where'.

In the model there are 'Absence' and 'Precedence' constraints.

For Absence(ApplyForTrip, 1) the corresponding code is following:

```
fact { lone te: TaskEvent | ApplyForTrip in te.assoEl }
```

This means, that there are zero or one (lone keyword) Events linked to the ApplyForTrip task.

For Precedence(Approve application, BookMeansOfTransport) the corresponding code is the following:

```
fun existsInBefore(currentEvent: TaskEvent, asso: AssociatedElement) : set TaskEvent {
        { hte: TaskEvent | hte.pos < currentEvent.pos and asso in hte.assoEl }
}
//    ensure sequence(Approve application, BookMeansOfTransport)
fact { all hte: TaskEvent | BookMeansOfTransport in hte.assoEl implies #existsInBe-
fore[hte,ApproveApplication] > 0 }
```

Here, we have a function, which takes an event and a task (AssociatedElement), and return the set of events corresponding to the given task happened before the given event. The constraint itself ensures that for each occurrence of BookMeansOfTransport, ApproveApplication happened before more than zero times.

What we have in this implementation is a log generator with limited amount of constraints and some potentially useful functions which were defined in Alloy code but were not used in the tool. Also we have some types of data like roles and resources (not shown here, see [1]). Finally, we have a working binary executable (without source code) for log generation, which proves that it is possible to use Alloy for this purpose.

There are several problems and limitations in the current implementation. Performance: the generation of 1000 traces with length up to 80 events (and average length is lower) takes around one hour (see original paper for measurements). Generalization: the current tool presented by the authors is made specifically for their model (does not have an input for it, model is hardcoded). Generation parameters: Only maximum trace length and amount of traces can be altered. Constraints and data: a limited amount of constraints is supported. Data represented as role and resource.

To sum up, the presented implementation is able to deal with specific type of constraints on data, which do not cover the whole spectrum of MP-Declare and it presents other performance and expressiveness limitations. In this paper we would like to overcome these limitations by proposing a solution able to generate a log from all types of the MP-Declare constraints. Also, we need to define some input format for MP-Declare model, and automate its conversion to Alloy.

6.2 Alloy: our proposal

Now we show how the baseline approach can be improved and modified to fit our purposes.

Trace. Our trace is a set of events, and each event has a reference to one activity.

```
abstract sig Event {
       task: one Activity
}

one sig TE0 extends Event {}
one sig TE1 extends Event {}
one sig TE2 extends Event {}
one sig TE3 extends Event {}

abstract sig Activity {}
one sig ApplyForTrip extends Activity {}
one sig ApproveApplication extends Activity {}
one sig BookTransport extends Activity {}
…
```

In this representation the length of the trace is fixed. To represent a trace of unknown length we could use ’lone’ multiplicity like this:

```
one sig TE0 extends Event {}
…
lone sig TE3 extends Event {}
…
```

However, it takes a lot of time to process this in cases where the difference between minimum and maximum values is too high. Therefore, in order to be able to represent traces of different length more efficiently, we will use special activity called ‘DummyActivity’, which may appear only in optional events, and will be removed from final trace. For example, if Alloy produce trace ‘A B DummyActivity C’, then it will be saved as ‘ABC’.

```
one sig DummyActivity extends Activity {}
…
one sig TE3 extends Event {} {not task=DummyActivity}
one sig TE4 extends Event {}
…
```

Finally, in all constraints we need order between events for two things: to see if one task is before/after another (Response, Precedence, etc.) and to see if one task is next to another (ChainResponse, etc). For this purpose, two predicates (Boolean function in Alloy) can be created:

```
pred Next(pre, next: Event){pre=TE0 and next=TE1 or pre=TE1 and next=TE2 or pre=TE2 and
next=TE3 or pre=TE3 and next=TE4 }
```

```
pred After(b, a: Event){// b=before, a=after
b=TE0 or a=TE4 or b=TE1 and not (a=TE0) or b=TE2 and not (a=TE0 or a=TE1) or b=TE3 and a=TE4}
```

This is an example for a trace of maximum length 4, but it is possible to write them for any length. An easier option would be to assign a number to each event, and use the comparison operators, but this would imply increasing the bitwidth with the length of trace, which is undesirable.

Constraints conversion. To constrain the Alloy model the 'fact' keyword can be used. All the constraints specified in fact blocks will be true in the generated solution.

In order to encode an arbitrary MP-Declare model to Alloy, we need to ensure that for each standard Declare constraint exists at least one possible way to encode it in Alloy (with existing trace definition). Some of the conversion rules are given in table 3:

Constraint	Alloy code
Init[A]	`taskA = TE0.task`
Existence[A]	`some te: Event | te.task = A`
Absence[A]	`no te: Event | te.task = A`
Exactly[A,N]	`#{ te: Event | A = te.task } = n`
Choice[A,B]	`some te: Event | te.task = A or te.task = B`
ExclusiveChoice[A,B]	`some te: Event | te.task = A or te.task = B` `(no te: Event | A = te.task) or (no te: Event | B = te.task )`
RespondedExistence[A,B]	`(some te: Event | A = te.task) implies (some ote: Event | B = ote.task)`
Response[A,B]	`all te: Event | A = te.task implies (some fte: Event | B = fte.task and After[te, fte])`
AlternateResponse[A,B]	`all te: Event | taskA = te.task implies (some fte: Event | taskB = fte.task and After[te, fte] and (no ite: Event | taskA = ite.task and After[te, ite] and After[ite, fte]))`
ChainResponse[A,B]	`all te: Event | A = te.task implies (some fte: Event | B = fte.task and Next[te, fte])`

Table 3 – Declare constraints encoding in Alloy

The full implementation is available in GitHub, and can be found here https://github.com/darksoullock/MPDeclareLogGenerator .

Here we explain some of the constraints presented above in more details.

Absence[A]

```
no te: Event | te.task = A
```

This can be read in the following way: There is no Event te ('te' stands for 'task event') where its task is equal to A.

Exactly[A,n]

```
#{ te: Event | A = te.task } = n
```

This one uses the cardinality operator '#', which means count or amount. The amount of Events such that task = A should be equal to n.

Response[A,B]

```
all te: Event | A = te.task implies (some fte: Event | B = fte.task and After[te, fte])
```

For all events where task is A there exists another event (named fte) whose task is B and which happened after A (after event te).

Adding data. Usually, in any process there is more information available about activities than just a name and a timestamp. In a XES log this information appears as additional attributes attached to events. The task of the log based MP-Declare generator is to provide a way of describing this data by specifying the type, possible values and additional constraints applied on this data.

The data constraint will be specified as additional function(s) on standard Declare constraints.

Based on examples and proposals in [4] and [7], we decided what data types can be used in event's data, and how they can be constrained.

The data types are the following ones:

Type	Description
Enumeration	The event attribute can have one of the values specified in the data definition.
Integer	The event attribute can have a value within the range of integers.
Float	The event attribute can have a value within the range of real numbers.

Table 4 – Data types supported in MP-Declare

We can use numbers in our solutions. However, numbers will affect the efficiency of the tool. Indeed, Alloy uses a SAT solver inside, and the numbers support is simulated through enumeration (not the most efficient way). Therefore, when running Alloy, one needs to specify a bitwidth, which defines the number of bits used in all int values. Then ALL the possible integers will be enumerated in signatures. This creates noticeable performance impact. The maximum practical to use bitwidth is 7 (possible values within the range -64..63). This is the main reason why it is not possible to add date/time value as a data type. This is also one of the reasons of the poor performance of [1] – in their solution Alloy computes indices of activities in traces from declarative specification.

Another problem with numbers is overflow. If bitwidth is 5 (possible values are -8..7), then cardinality operator #(Task) for 8 tasks will return -8, which can lead to wrong evaluation of constraints. For example, if we have a constraint Absence[A,2], then Alloy code #(A) < 3 will also evaluate to true when A occurs 8 or more times.

Based on this, we restricted the usage of numbers in our solution (will be discussed below).

Each Declare constraint can have one or two data constraining function, depending on number of constraints. Examples:

```
Existence[BookTransport B] | B.Price>50
```

This constraint has one function and means ‘Activity BookTransport with Price>50 should present in a trace’.

Another example from [23]: ‘a bank account is opened only in case risk is low‘. This can be written as:

```
Precedence[AccountChng Acc, RiskEval Risk] | Acc.Status is Opened | Risk.Level is Low
```

This means, that when event ‘account changed’ occurs with status ‘opened’, it has to be preceded by the event ‘risk evaluated’ with ‘level’=’low’.

The operators thet can be used to constraint data are:

Name	Description	Scope	Example
is	Value is equal to something	`Enum`	`A.Transport is Car`
is not	Value is different to something	`Enum`	`A.Transport is not Car`
in	Value is equal to one of (…)	`Enum`	`A.Transport in (Car, Train)`
not in	Value is not one of (…)	`Enum`	`A.Transport not in (Car, Train)`
or	One of the arguments should hold true	`op`	`A.Transport is Car or A.Transport is Car`
and	Both arguments should hold true	`op`	`A.Transport is Car and A.Transport is Car`
not	Negation	`op`	`not A.Transport is Car`
same	For constraints with two arguments (e.g. response). ‘same X’ means A.X equals B.X	`Enum, numbers`	`same Transport`
different	‘different X’ means A.X not equals B.X	`Enum, numbers`	`different Price`

>
<
>=
<=
=
	Comparison of variable with constant. Comparison of two variables is not supported (e.g. A.Price>=B.Price)	`Numbers`	`A.Price>50`
			
			`Incorrect: A.Price<B.Price`

Table 5 – Data constraints operators

These operators can be combined in different ways (nested or, and, not). To ensure the correct order of constraint evaluation, parenthesis can be used. The priorities of the operators without parenthesis are the following: not, and, or. The empty function for a data constraint will be evaluated as true.

To encode the data in Alloy code we introduced the ‘Payload’ signature.

```
abstract sig Payload {}
```

And added in each event set of payloads.

```
abstract sig Event {
	task: one Activity,
	data: set Payload
}
```

When we want to add a new datatype, we create anabstract signature inherited from payload. This signature will be the name of the new type (attribute in the log). From this signature we create more subsignatures, which will be the values.

For example, if we want to create the data type ‘TransportType’ with possible values ‘Car’, ‘Train’, and ‘Plane’, the code will be:

```
abstract sig TransportType extends Payload {}
fact { all te: Event | (lone TransportType & te.data)}
one sig Car extends TransportType{}
one sig Plane extends TransportType{}
one sig Train extends TransportType{}
```

In the second line we define the constraint indicating that one event can have at most one instance of this type of data (e.g., an event cannot have both: car and train).

The next step is to bind the data to activities – define which type of data can be used in which events. Suppose that we want our TransportType data to be attached to Book-Transport events. We write:

```
fact { all te: Event | te.task = BookTransport implies (one TransportType & te.data) }
fact { all te: Event | some (TransportType & te.data) implies te.task = BookTransport }
```

This code ensures that for every BookTransport event a TransportType is attached and vice versa – the activity of an event with TransportType attached to it can only be BookTransport.

In case of multiple data types bound to one event, or one type used in different events, the constraint is similar. The following example shows the binding of two activities Book-Transport and Use transport to data type TransportType

```
fact { all te: Event | te.task = BookTransport or te.task = UseTransport implies (one
TransportType & te.data) }
fact { all te: Event | some (TransportType & te.data) implies te.task = BookTransport or
te.task = BookTransport }
```

As Alloy does not really support big numbers, they are mapped to some intervals and written as enumerated values. This mapping is performed before running the Alloy analyzer in the pre-processing step. After a solution has been found, the values corresponding to numbers are unmapped. At this point, hence, each numeric variable is not associated to a single value but rather to an interval of admissible values. A random value within the admissible set of values is hence picked and assigned to the variable in the post-processing step.

Data constraints encoding in Alloy. As mentioned before, constraints containing data will be the same as ordinary Declare constraints, but will include functions for data. Alloy already has all the necessary operators for implementing the following operations: is, in, not, or, and (all operations not related to numbers).

Consider the example:

```
Precedence[AccountChng Acc, RiskEval Risk] | Acc.Status is Opened | Risk.Level is Low
```

We can write predicates (Boolean functions) that take the event as a parameter, and return if the condition is true. For 'Acc.Status is Opened' it will be:

```
pred p1 (A: Event) { { A.data & Status = Opened } }
```

Predicate P1 will return true if the intersection of the data attached to the event and the set of possible statuses (this intersection will be always one value, as constrained before) will be equal to 'Opened'. Similarly, for 'Risk.Level is Low' the predicate will be:

```
pred p2 (A: Event) { { A.data & Level = Low } }
```

Therefore, to encode original constraint, we first encode the constraint without data.

```
Precedence[AccountChn, RiskEval]
```

The corresponding Alloy code will be:

```
all te: Event | te.task = AccountChn implies (some fte: Event | fte.task = RiskEval and
After[fte, te])
```

And then we can add data constraints to it by calling the predicates defined above:

```
all te: Event | (te.task = AccountChn and p1[te]) implies (some fte: Event | fte.task =
RiskEval and After[fte, te] and p2[fte])
```

Note, that in Alloy arguments of functions and predicates are passed in square brackets.

Data can be added in the same way also to other Declare constraints.

For the 'same' and 'different' operators we need access both, activation and correlation activities, therefore the second function will have two arguments. An example of predicate for the 'same Level' operation is:

```
pred p3 (A, B: Event) { { A.data & Level = B.data & Level } }
```

Handling numbers. Suppose we have an activity with associated price. Price should be a number. Also suppose that we have a constraint assessing that price should be greater than 50.

```
Existence[Activity]|Price>50
```

To implement this in Alloy we first we find all the numbers involved with this data type (price). In our case there is only one value – 50 ('in the expression Price>50'). Then we split all numbers into intervals. In this case we have two intervals: more than 50 and less or equal to 50. Finally, we add a constraint in Alloy expressing the fact, that the value of price should be in the valid interval (in this case >50).

```
some te: Event | te.task = A and Price = IntervalGreaterThan50
```

Afterwards in the post-processing, the interval will be replaced by a random value within the interval.

If there are more constraints involving different numbers, they will be split into more intervals, and Alloy will try to find one, that satisfies all of them.

Example: We have two constraints:

```
Absence[SomeTask A]|A.price>100
```

```
Absence[SomeTask A]|A.price<50
```

(data definition omitted). In these constraints, price is compared with two numbers: 50 and 100. Based on this, the following intervals are generated: <50, >=50&<=100, >100. Then

they are encoded in Alloy as signatures: LessThan50, Between50And100, MoreThan100. The first constraint (A.price>100, converted to 'A.price in (MoreThan100)') allows only the LessThan50 and Between50And100 intervals, and the second constraint allows the Between50And100 and MoreThan100. Therefore, in the Alloy solution the price will be equal to Between50And100. This value is then replaced by a random number between 50 and 100.

'same' and 'different' data constraints on numbers. With intervals representing numbers in Alloy we cannot use 'same' and 'different' constraints in the same way as we do for enumerated values. Indeed, if two values are within same interval, they might have same or different values (unless the interval has only one possible value, e.g., an integer between 1 and 3 can only be equal to 2). If we constrain two values to belong to different intervals (e.g. LessThan50 and MoreThan50), they will surely be replaced with different values, but in many cases this is not possible.

Our solution is the following: add tokens to events (these tokens are not related in any way to tokens in Petri Nets). We encoded in Alloy following rules for the tokens:

- Each token has a matching pair. Like poles of the magnet, one cannot exist separately.
- Each token references data attribute of an activity, and the same attribute cannot be referenced from both token in pair.
- If two numeric data attributes in two activities are referenced from pair of tokens, the value of these attributes will be the same interval (only for 'same' constraint).

If two events have the same token, they will be guaranteed to get the same value in the interval of numeric data attribute, and during post-processing we will ensure, that the same random value will be selected for both attribute. The implementation of the tokens for 'same' and 'different' constraints is identical in Alloy, only constraints themselves differ.

The final definition of event, which includes tokens is following:

```
abstract sig Event {
	task: one Activity,
	data: set Payload,
	tokens: set Token
}
```

And the tokens definitions are following:

```
abstract sig Token {}
abstract sig SameToken extends Token {}
abstract sig DiffToken extends Token {}
```

For example we have following constraint:

```
Response[BookTransport, UseTransport]||same Price
```

Activity BookTransport should always be followed by activity UseTransport with the same value of the data attribute Price. To achieve this, we add a pair of tokens which will reference these two activities, and they will get the same interval for Price. Then in post-processing step we will know, that two activities with the same token should get exactly the same value in the corresponding data attribute (in this case it is Price).

As we will handle tokens outside Alloy, we need to ensure that two values cannot be the same and different simultaneously. This also includes indirect relations like A=B, B=C,

A≠C. The encoding of such transitive relation would significantly affect the Alloy performance (and could be complex, as recursion is not supported). To deal with this, we decided to restrict each data variable with either ‘same’ or ‘different’ token type with the following code:

```
fact { all te:Event| no (te.tokens & SameToken) or no (te.tokens & DiffToken) }
```

Though it may decrease the amount of possible traces, the models that are affected by this limitation are quite uncommon (we did not encounter any in related literature or real life models). A possibility for solving this type of limitation could be spliting each interval into two or more parts (it will make ‘different’ token is no longer necessary).

Consider the example with two constraints:

```
X: integer between 0 and 100
Response[A,C]||different X
ChainResponse[A,B]||same X
```

Possible trace would be A[X=1]B[X=1]C[X=2]

If X consists of one interval, a solution will not be found, as events A and C need ‘different’ tokens, and events A and B need the ‘same’ tokens; first event need to have tokens of both types (which we do not allow). But if we have two intervals, then events A and B will have ‘same’ tokens, and event 3 will be in different interval, therefore nor requiring ‘different’ token, as different value now guaranteed even without it. You can look at ‘testSameDifferentForOneEvent’ unit test in code, to see that it is work.

As we see from the Event definition, tokens are not explicitly bound to data. Therefore, we use signatures’ names for distinguishing types and constraining the tokens usage in Alloy facts. For example:

```
abstract sig SamePrice extends SameToken {}
one sig SamePrice1 extends SamePrice {}
fact {
all te: Event | SamePrice1 in te.tokens implies (one ote: Event | not ote = te and SamePrice1 in ote.tokens)
}
```

The meaning is the following: if a token is present in event, it must appear only once in other event. Thus we always have no or two usages of token, hence it will not appear in random events, obscuring the model. Also, it will not persist the same value in multiple activations of constraint (e.g. if we have ‘ChainResponse[A,B]||same X’ and trace ABAB, X will not necessarily have the same value in all four events, but will rather have two different pairs).

In this example we had only one non-abstract token, but in practice we have more. Otherwise, if model will contain two activation of constraint, there will be not enough tokens for the second activation.

The number of instances of some signatures in Alloy should always be bounded. In this case we use ‘one’ multiplicity. Without this bound we would need to specify the scope (maximum occurrences of signature in the model) when running the analyzer. This would weaken our control over individual tokens (i.e. made post-processing logic more complex), and would add complexity to evaluate required scope (lower performance).

The described idea work for simple cases, but as we can use arbitrary expressions, some of them may produce unwanted behavior. Suppose that we have ‘not same A’. For this expression generated code will look for something like ‘not (equal intervals and token present)’ which is the same as ‘not equal intervals or not token present’. This may lead to a broken model, which has different intervals with same token, and it will be discovered only in post-processing. The solution for this problem is very obvious: replace all occurrences of negates same to different and vice versa. This should also take into account cases, where negation is indirect like ‘not (A and same B)’. This approach was implemented in the presented solution and all the details can be found in the source code. https://github.com/dark-soullock/MPDeclareLogGenerator

Another similar problem occurs when we want to specify negative constraints, like NotResponse. Suppose we have ‘NotResponse[A,B]|| same X’. To make it work, we can encode it like ‘A should be not followed by B or A followed by B with different X’. As we have numbers and ‘different’ constraint implemented with intervals and tokens, it will be encodet following way: ‘if A present then it is not followed by B with same interval and without “different” token for X’.

In Alloy terms the function ‘same X’ for this constraint looks like this:

```
{ A.data & X = B.data & X and (not ( one (DiffX1 & A.tokens & B.tokens))) }
```

Consider following model:

```
X: integer between 0 and 100
RespondedExistence[A,C]||same X
RespondedExistence[B,C]||same X
```

Our algorithm will iterate over tasks. A should have the same value as C, but as C does not have value yet we assign a random one. Suppose it is 1. The same happens for B, we assign a random value, suppose it is 2.

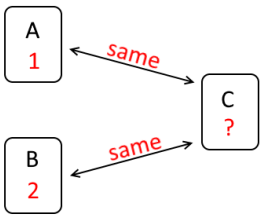


Figure 4 – Deadlock example when assigning values in post-processing

Now we need to assign a value to C, but both A and B have a value, and they are different. Therefore, we have an invalid state in our program. To avoid this, we find all the groups of connected events (events which should all have the same value for a data attribute), and add tokens for all pairs within each group. In this example, we would add matching tokens in A and B, and this would lead to assigning the same value for B as for A.

For ‘different’ constraint, this problem does not occur, because if A different from B, and B is different from C, then A can be the same as C.

As mentioned in the background, traces have both trace and event attributes. The Alloy-based log generator is able to support both of them. However, while for the event attributes the log generator is able to completely constrain them, for trace attributes, it only allows for constraining admissible values for enum and numerical variables. Moreover, they are not part of the Alloy model, and will be handled exclusively in java code (on pre- and post-processing stages).

The entire procedure of log generation is following:

1. Find all numeric data definitions in input
2. For each of them find all usages in comparison operations and save numbers it compared with
3. Split space of possible values in intervals by found numbers
4. Map each interval to codename
5. Generate signatures for Alloy
6. Run the Alloy analyser
7. Unmap all codenames from the result back to intervals
8. Replace intervals by one random value from it

In this flow, step 1 is performed by model parser (Fig 3.1.1 – generator architecture), steps 2, 3 and 4 are performed in pre-processing, 5 is code-generation, and 7..8 done in post-processing.

7 The NuSMV-based log generator

In NuSMV a model is stored in an smv file. It has several blocks. The VAR block contains all variables. Each variable present in all states of the trace. We will use four types of variables: Boolean, enumeration, integer and real. FROZENVAR stores constants – variables which do not change their value over states. In the ASSIGN block, we can define the possible values for the variables. We will use three constructions in this block: init(var) – assign initial value to the variable; next(var) – define in which way the value can change. LTLSPEC block – here we encode constraints as LTL formulas.

To produce a trace, we generate a counterexample, negating all constraints. Consider 'ChainResponse' example.

Suppose we have an activity variable, which stores one of three values {a,b,c}. We want to add a constraint ChainResponse(a,b). In LTL it will be G (activity = a -> X activity = b). It means, that for all states if the current activity is a, next should be b. Now we need to negate it. I.e. we consider !G (activity = a -> X activity = b), and find a counterexample for it. The counterexample will provide us with an execution trace for the non-negated (original) model.

Figure 6.1.1 shows translation of standard Declare constraints to negated LTL:

Constraint	!LTL
AlternateResponse(a,b)	!G (state = a -> X(state != a U state = b))
Response(a,b)	!G (state = a -> X F state = b)
Existence(a)	!F (state = a)
Absence(a)	F (state = a)
Choice(a,b)	((!F state = a) & (!F state = b))
ExclusiveChoice(a,b)	((!F state = a) & (!F state = b) | (F state = a) & (F state = b))
RespondedExistence(a,b)	((F state = a) & (!F state = b))
ChainResponse(a,b)	!G (state = a -> X state = b)
NotRespondedExistence(a,b)	((F state = a) & (F state = b))
NotResponse(a,b)	!G (state = a -> X !F state = b)
NotChainResponse(a,b)	!G (state = a -> X state != b)
Precedence(a,b)	(state=a V state!=b)
NotPrecedence(a,b)	!G (state = b -> X !F state = a)
NotChainPrecedence(a,b)	!G (state = b -> X state != a)

AlternatePrecedence(a,b)	!G (state = a -> (X(state != a U state = b))) | (state=a V state!=b)
ChainPrecedence(a,b)	!G(X state = b -> state = a)

Table 6 – Encoding constraints in LTL

Init, ExistenceN, ExactlyN and AbsenceN are not shown in Figure. To encode Init(a) we write “init(activity):=a;” in the ASSIGN block. There are two options to encode Existence(a,N), Exactly(a,N) and Absence(a,N): we can use a counter:

```
VAR
        state : {a, b};
        aCount:integer;
ASSIGN
        init(aCount):=0;
        next(aCount):=
                case
                        state= a: aCount +1;
                        TRUE: aCount;
                esac;
```

Then we can add LTL constraints to this counter to keep it within bounds.

Alternatively we can express them in LTL. If we want to encode Existence(a,2), we use the negated formula !F (activity=a & F (activity=a)). Similarly, we can encode Absence. Exactly(a,N) can be written as Existence(a,N) & Absence(a,N).

In order to add data to the model, we define additional variables in the VAR block, and then data constraints can be attached almost as-is to the relevant part of LTL. Example encoding of two data attributes:

```
VAR
        transportType : {car, bus, train};
        price : integer;
```

NuSMV generator supports all the data constraints allowed in our MP-Declare model except ‘same’ and ‘different’.

NuSMV produces infinite traces. In order to create a finite trace out of an infinite one, we will add at the end an infinite tail with a dummy activity. In particular the following LTL rule is used: "! F(activity = _tail) | !G (activity = _tail -> X activity = _tail)". In Declare terms this is Existence(_tail) and ChainResponse(_tail, _tail). Once it start it will never end, and therefore we keep only part of the trace, which is located before the first occurrence of _tail.

Each run with the same model in NuSMV produces the same counterexample. In order to produce different traces, we can disallow in the model all previously generated traces. We don’t need to bind data to activities, because each state has all the variables (as if all activities were bound to all the attributes). Instead, we will remove variables from irrelevant events in post-processing.

Suppose we have a trace abc. The LTL rule “first & activity = a & X (activity = b & X (activity = c & X (activity = _tail)))“ will disallow this trace. “first” should be the Boolean variable, which is true only in the initial state (easily implementable in the ASSIGN block).

NuSMV always generates the shortest trace possible. To allow generation of a log with traces of different length, we generate several smaller logs with fixed length of traces between minimum and maximum and then join them into one. For example, if we need a log with 1000 traces of length between 20 and 24, we generate 5 logs with 200 traces for each length, and then join them. This behavior also supported in Alloy generator as an additional option.

The NuSMV generator is also available on github. It has two parts: .smv code generator is a module in the repository with Alloy generator (because they use a common module for parsing the MP-Declare model) https://github.com/darksoullock/MPDeclareLogGenerator. Second part is generator itself and located at https://github.com/darksoullock/SmvToXes. It also requires NuXMV binary, which can be downloaded separately at https://nuxmv.fbk.eu/ (it cannot be included in repository due to license).

8 Implementation

There are three options to run the generators: API, command line and Rule Mining Web App (RuM).

There are 5 mandatory parameters required to run the generators: minimum trace length, maximum trace length, amount of traces to generate, MP-Declare model, name of the output file. Optional parameters are vacuity, noise (amount of negative traces), and whether uniform distribution of trace lengths required.

Both of our generators support options for vacuity and noise. When vacuity constraint is enabled, all the constraints in the input model will be activated at least once in each trace of the log. The noise in the log is represented by negative traces – traces which have at least one of the constraints violated.

Furthermore, Alloy has additional parameters:

- constraints shuffling – which allows the generator to improve variability by reordering priority of the constraints during generation
- split of intervals – allows for the generation of more traces for overconstrained models
- reuse of Alloy solutions – allows to get more traces for models with numeric data attributes
- names encoding – allows to use spaces and special characters in names of activities and data attributes
- maximum ‘same’ instances, do not request tokens for single-value interval – options which allow to balance performance and log variability in some very rare cases. Usable only for models with ‘same’ or ‘different’ constraints on numeric data attributes.

API is the most convenient way to use the generators from another application (if stack of technologies matches). It is represented by exported functions from library. The Alloy-based application is a jar package, and the NuSMV-based is a CLR assembly. The API is used internally in the command line interface, and in the RuM plugin.

When the command line interface is started without arguments it shows a help page with the description of the parameters.

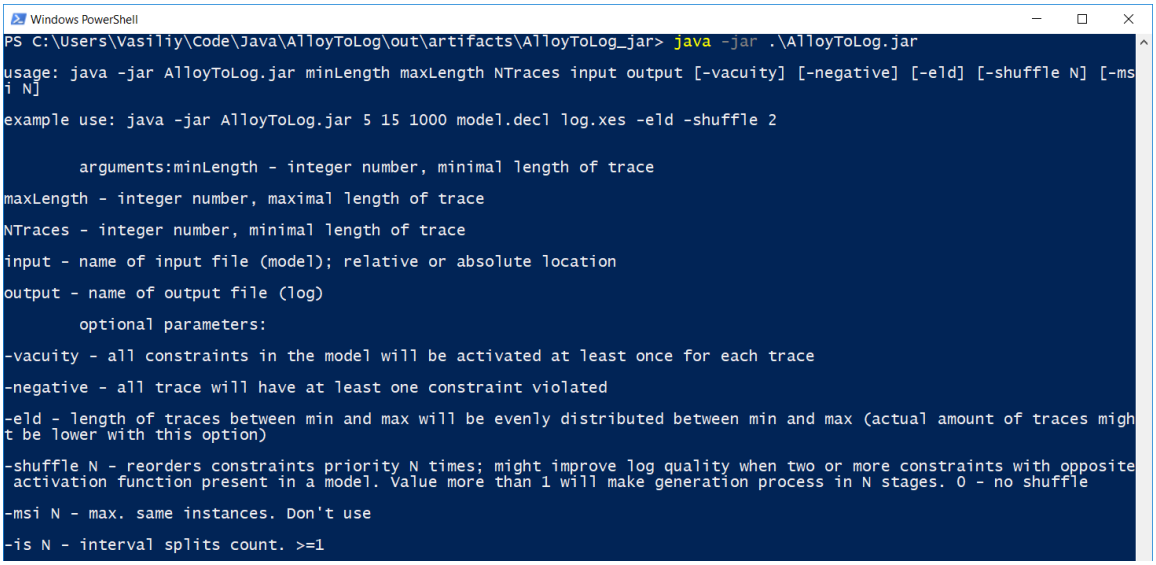


Figure 5 – Alloy-based generator CLI

RuM is a web application, that contains different tools for process mining as plugins. We have implemented two plugins for it: a model designer and the generator itself. The model designer allows the user to specify in a graphical interface an MP-Declare model (that can be given as an input to the log generator) without knowing the syntax of the input language. It also helps to avoid some errors (like typos in name of activity) and contains description of constraints. The model is stored in the proposed textual format

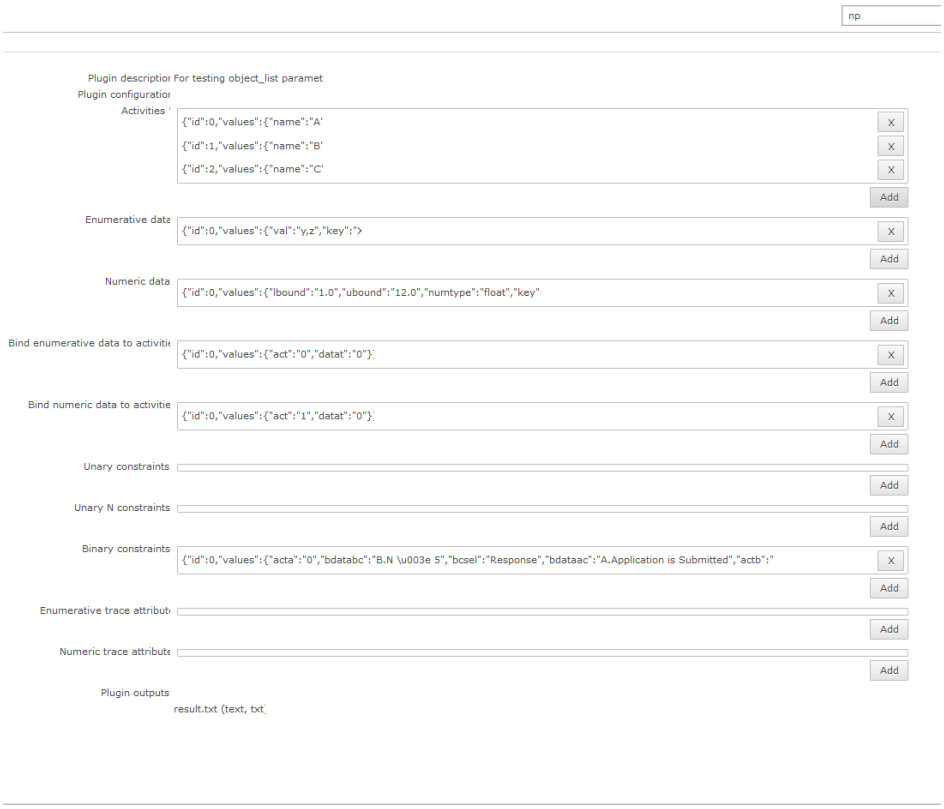


Figure 6 – Adding data constraint in RuM

Figure 6 shows the dialog window for adding MP-Declare constraint.

Figure 7 shows the RuM interface for the designer plugin opened

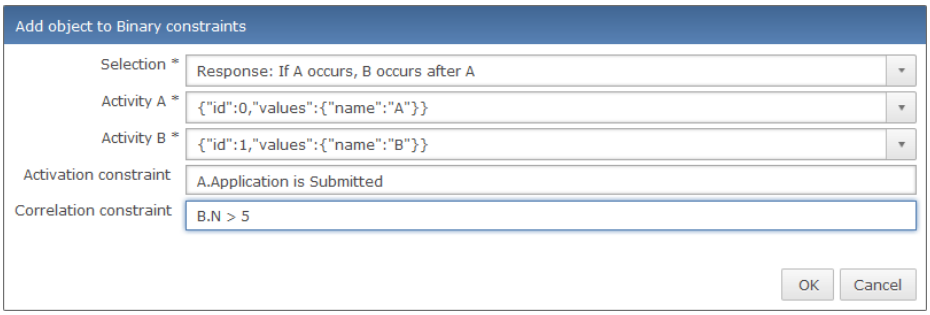


Figure 7 – Declare designer in RuM interface

The designer allows the user to specify the list of activities, the data attributes (enumerative and numeric separately), the data binding, the constraints and the trace attributes to create an MP-Declare model

The second plugin is the generator itself, where the generation parameters including the input model can be specified.

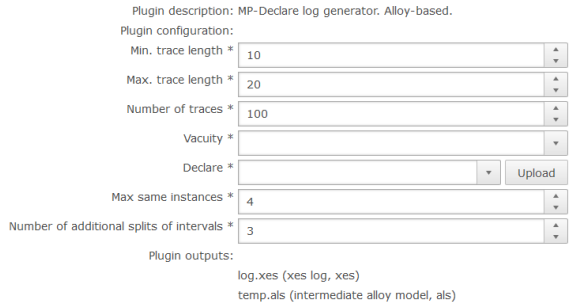


Figure 8 – Log generator in the RuM interface

The application runs all the tasks on a server backend, and stores them in the user's projects. Therefore, when the generation is started, the user can close the browser tab and return to it later to download the log or check the status.

9 Log quality evaluation

In order to compare and improve the generators, we need a way to evaluate the quality of the generated logs. For this we need to find what properties make a log good or bad. One of the important features of a good log is that it contains variegated execution paths. This means that in different traces different constraints are activated, activities occur in different order, etc.

To find a metric to measure the quality of a log, we looked at the literature related to a trace clustering. Though we don't need to cluster traces, clustering algorithms require a function which defines the distance between two objects (traces in our case). We can try to use these metrics for comparing traces and get a number, which shows how different are traces in the given log.

In [13] the authors propose to use so-called profiles to convert a trace into the vector of integers, and then use distance metrics to compare these vectors. For example, an activity profile uses one dimension in the vector for each activity in a process. Its value will be equal to the number of occurrences of the activity in the trace. For instance, if a process has activities <a,b,c,d>, then for trace "aaacbc" the activity profile will be the vector <3, 1, 2, 0>, i.e. the amount of the corresponding activities in the trace. The transition profile – similar to the activity one, contains all possible combinations of two activities. Each pair (a, b) stands for transition a->b. The values in the vector are based on the amount of transitions. For example, if a process has activities <a,b,c>, then for the trace "aaacbc", the transition profile has the following values <2, 0, 1, 0, 0, 1, 0, 1, 0> corresponding to the frequencies of the transitions {a->a, a->b, …, c->c}. As we have data attached to activities, we also need to evaluate their variability. There are two profiles for data: case attributes and event attributes. Then we count the amount of occurrences of each data attribute in a trace. The difference between these two profiles is, that for 'case' we count data attributes of the case, and for 'event' we distinguish between the same attribute attached to different events in the same case. For example, suppose we have two activities A and B with data attribute X attached to both of them. Then in 'case attribute' profile of trace 'AB' vector will be <2> for two occurrences of X in trace, and in 'event attribute' profile vector will be <1,1> for occurrences of X in activities A and B. The last profile 'performance profile' uses parameters of the traces, such as length, case duration, etc.

After transforming each trace into a vector we can compare each trace with all others using distance metrics (such as Euclidian, Jaccard, Hamming, cosine, or other heuristics), and then use the average distance as a log quality score.

With this approach we will consider the flow of events, and can compare traces of different length. Its disadvantage is that some different traces can get the same score, and therefore distance equal to zero (unlikely, though). Also we treat data separately from events, therefore same data attributes in different places will give the same values in vector.

In [14] the authors propose to assess the degree of similarity between two traces by considering subsequences of activities which are conserved between multiple traces. This approach is similar to k-grams. The introduced difference is that authors use not all possible subsequences of length k, but rather select only important ones (see the referenced paper for more details about which sequences are important). They count the amount of occurrences of each subsequence to transform a trace into the vector. However, rules for extracting features are quite complex and cannot be easily extended to use data.

In [15] the authors mention several options to get features from traces. Among them there are bag-of-activities (which does not capture order), k-gram model (which is computationally expensive), Hamming distance (only for traces of the same length), edit distance. We have designed our own metric based on edit distance, and specifically adapted to capture trace similarity, and to produce a score between 0 and 1.

The main idea of our Levenshtein-based edit distance metric (to compare two traces) is the following: in the first iteration we calculate the minimal amount of activities which can be inserted, deleted or replaced in one trace to obtain the other. For each of such modifications we add one to the distance score. Then, for each activity untouched at the first step, i.e. which is present in both traces and has the same data attributes, we count the amount of different data values, and add to the score this amount divided by the total number of data attributes (so it will be between 0 and 1). This will ensure that we are not comparing data attributes of unrelated activities. Finally, we divide our score by the length of the longer trace, and get a score value in the range between 0 and 1. For two completely different traces this metric will produce 1, and for identical traces 0.

Suppose we have traces "aab^{xy}ca" and "ab^{xz}aa", and that activity 'b' has two data attributes. At first we count the minimal amount of insertion/replacement/deletion operations. For these traces it is 2 – we can remove the first 'a' and replace 'c' with 'a'. We see, that 'b' is not affected by these changes, so we can add 0.5 (1 different attribute value divided by 2 total) to the score, and get 2.5. Finally, we divide it by the length of the longer trace (5), and get 0.5.

When we have a metric to compare the variability of two traces, we can calculate the variability of a whole log by adding distances between each pair of traces, and dividing this sum by their amount (N^2). So the output value will be within the same bounds as the metric value for two traces.

For testing our generators, we generated 20 logs of 250 traces each from the model M4. Difference between the logs is length of traces, which starts from 7 and is increased by 1 up to 26.

In Figure 9 we can see the scores obtained by each of the four metrics: first three of them are based on distance between vectors described above and in [13], and use Manhattan distance, Euclidian distance and sine distance (sin of angle between vectors) to compare vectors. The fourth one is the Levenshtein-based edit distance metric.

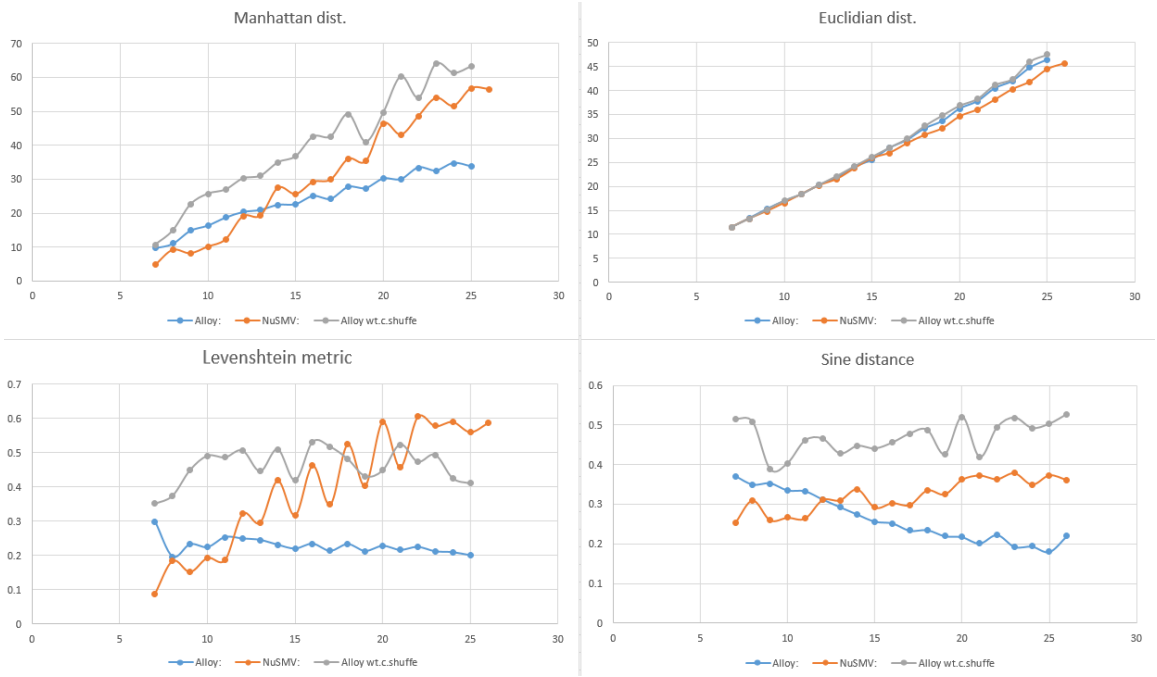


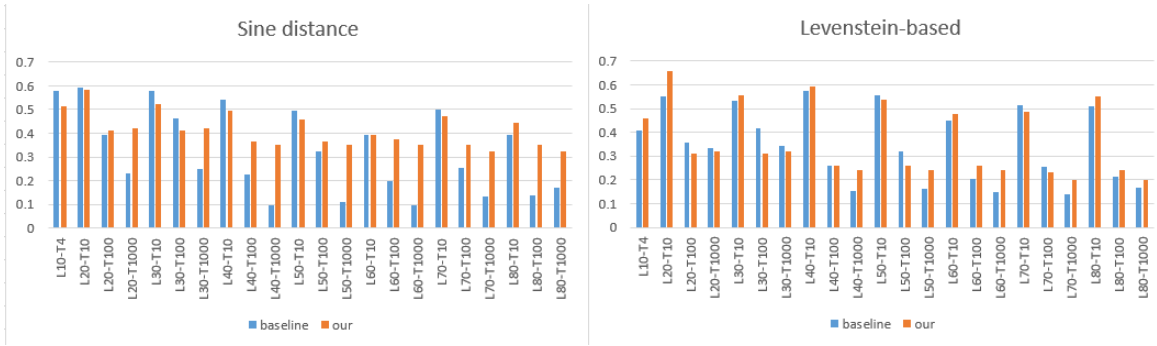
Figure 9 – Variability metrics for generators

On the X axis of these plots we have the length of the traces in the log. The values on the Y axis represent the variability score, where higher is better. The three lines represent values for NuSmv generator (orange), Alloy (blue), and Alloy with constraints shuffling enabled (see Subsection 6.2 and Section 8). In the next subsection a more detailed analysis about how this parameter can influence the variability of the generated logs will be presented.

As we see in the Figure, metric with Euclidian distance performs poorly (i.e. no relevant difference comes out when comparing the generators), so we will not include it in the future tests.

The difference in the results of the Levenshtein-based metric and the vector-based ones is that Levenshtein-based assigns scores to data variability only if data pertains to the same activity, whereas vector-based metrics treat data independently of activities. The Manhattan and sine distances are very similar, but sine has limited maximum value, and therefore can be used for comparing logs of different sizes.

We also compare the quality of the logs generated with our Alloy generator to the one of the logs generated by the approach presented in [1,2]. For this we recreated exactly the same model (M1) used in [1], and generated logs with the same amount and lengths of traces. After this we run our metrics on both logs. The results are shown in Figure 10.



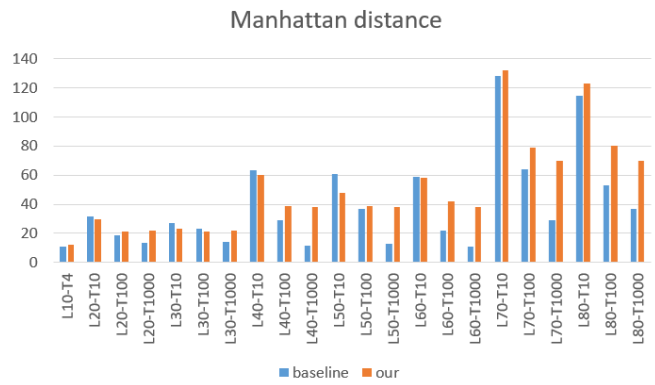


Figure 10 – Log variability comparison with baseline

As we can see, the scores are very similar for each log. Note, that we should not compare scores of different logs as they have different size, and therefore different probabilities of containing similar traces.

Unfortunately, we cannot do the same with NuSMV generator because of unsupported 'different' constraint used in model M1.

Constraints shuffling

The order of statements in the Alloy matters, and for different ordering of constraints in the model we will get different logs generated. Based on this observation, constraints shuffling was implemented. When this option is enabled, instead of generating N traces in a single run we generate N/S traces S times (S is an input parameter defining how many times we want to perform shuffling), randomizing the order of constraints at each run. As a result, we are getting more variable logs with unbiased priority of constraints activation.

Figure 11 and 12 show the results of the generation of 100 traces for two MP-Declare models: 'business trip' and 'loan application'.

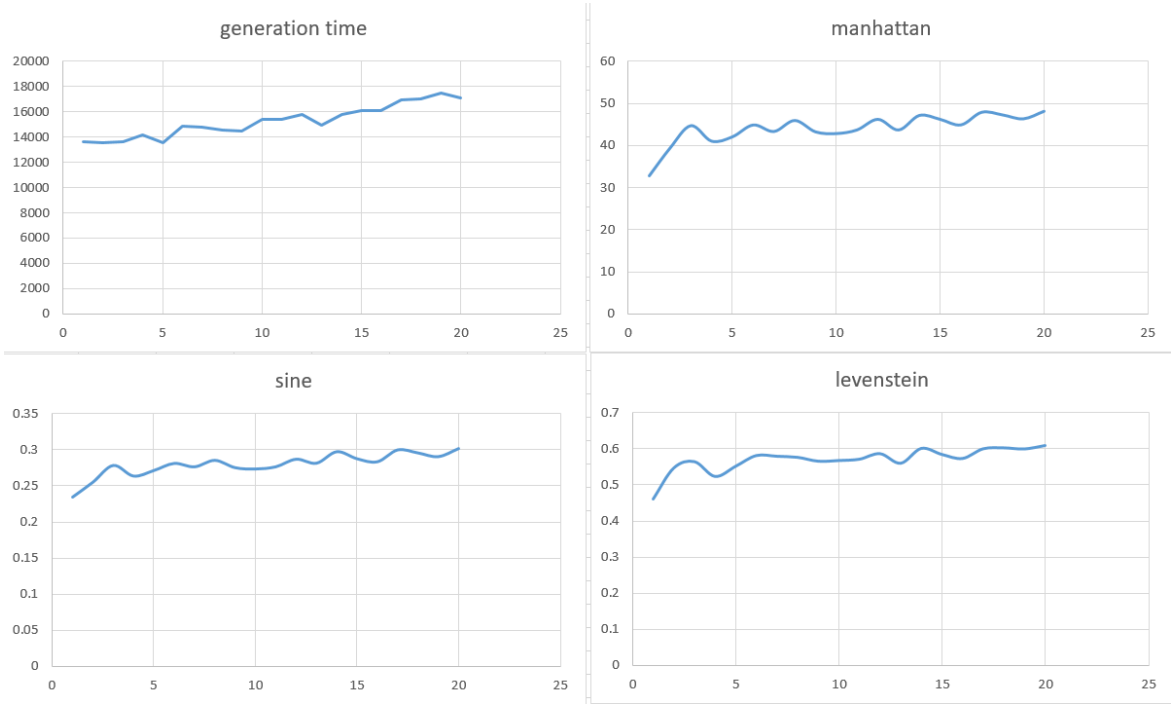


Figure 11 – Alloy constraint shuffling test for the loan application model.

On the X axis of all graphs there is S – how many times constraints were shuffled during the generation of the log. On the Y axis of the first plot we have the generation time. It increases from ~14 to ~17 seconds. In next three plots we have the variability scores measured using Manhattan and sine distance of vectors-based and our Levenshtein-based metrics. As we can see, S=2 and S=3 are improving the variability noticeably with very little impact on generation time, and S>=3 still tends to improve it further.

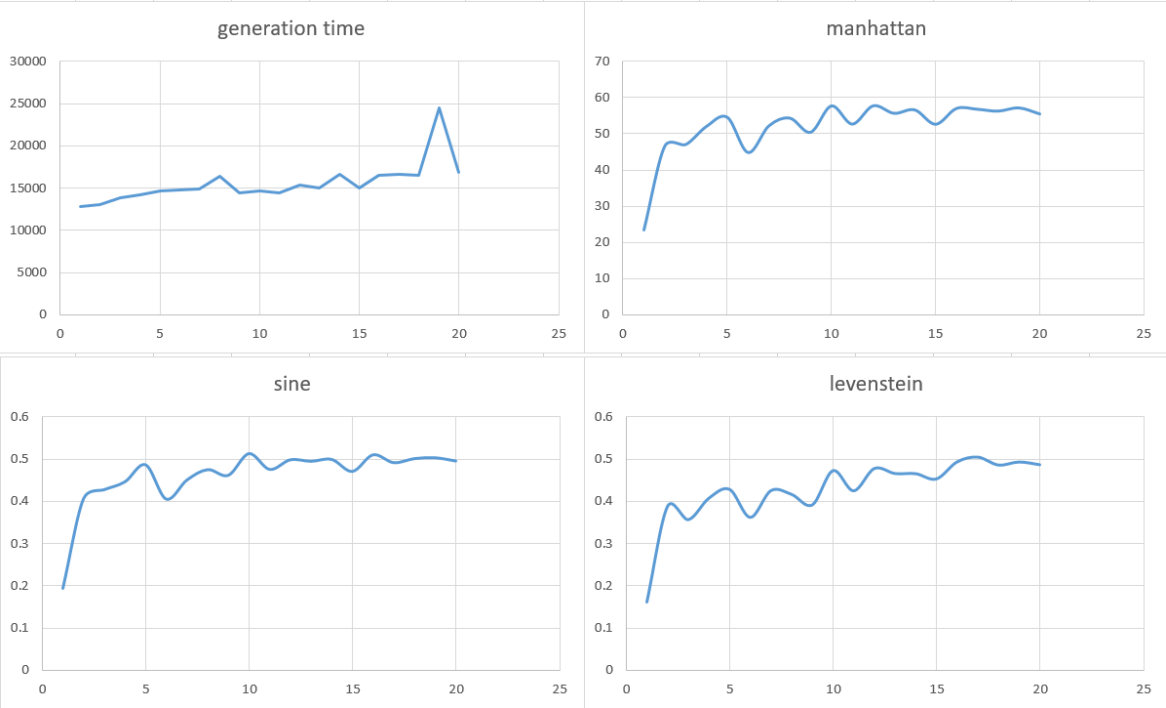


Figure 12 – Alloy constraint shuffling test for the business trip model

The same test was performed for the SMV-based generator to study, whether it has similar behaviour. The plots in Figures 13 and 14 show distance metric for logs, generated from the same two models as in the Alloy test.

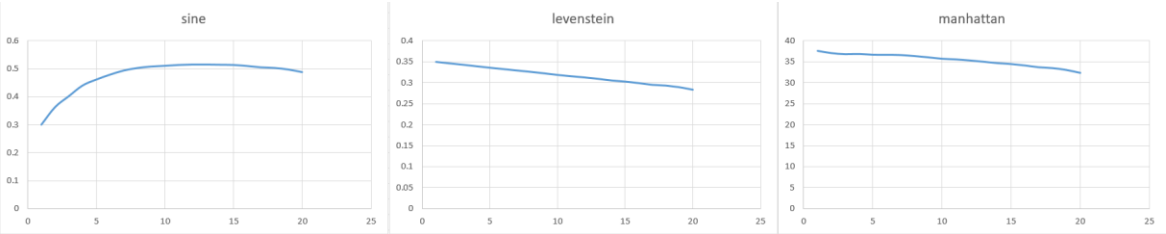


Figure 13 – loan application model constraint shuffling SMV test.

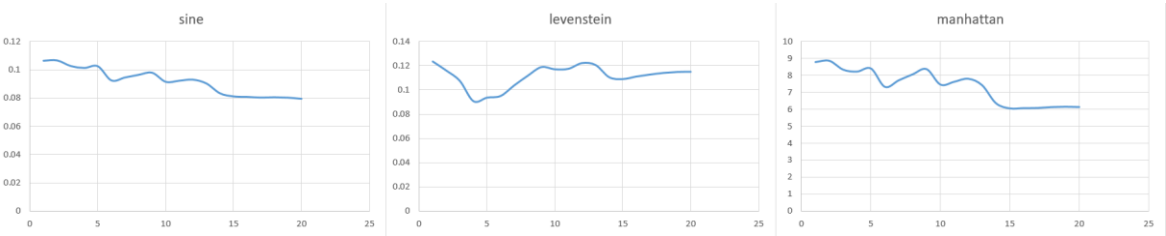


Figure 14 – business trip model constraint shuffling SMV test.

From these plots we can see, that shuffling variables/constraints order between trace generation does not help to improve the log quality, when using the SMV-based generator.

We noticed that regardless of the order of constraints and variables in NuSMV, the first produced trace is always the same. Also, with each restart of the generator all numeric data attributes are getting the same values, which has a negative impact on the data variability.

Evaluation of numeric data attributes randomness

In our variability metrics we treat all data attributes as enumerative, because use the considered metrics cannot compare two numbers. To fill this missing part we performed another test for measuring randomness of generated numbers.

Our test is based on one of the diehard tests [24] called “The count-the-1's test for specific byte”. In this test we count the 1 bits in last byte of each number. Then we convert the counts to "letters", and count the occurrences of five-letter "words". Finally, we apply Pearson’s chi squared test on them. As a result we get a single number, where lower value means better randomness.

We tested the randomness of numerical values on the business trip model. In this model we have two numeric data attributes: ‘Speed’ and ‘Price’. We use 20000 values of price attribute, and 10000 for the ‘Speed’ because it occurs less frequently. Next table shows the results.

	Price	Speed
NuSMV	2.06561064599396	4.06542653075637
Alloy	11.1843106956579	9.59869070406868
Alloy with shuffling	6.68381883718214	1.35307559528014

Table 7 – Randomness test on generated logs

In this test the Alloy-based generator performs worse than NuSMV. Though the Alloy-based generator assigns values in java post-processing (which should assure proper randomness), one of the constraints in this model is ‘RespondedExistence[BookTransport A, UseTransport B] | A.Price=50 | B.Price=80’. Therefore, we have two intervals consisting of a single value (50 and 80), which are selected with comparable frequency to other intervals, and always replaced with the same value. That means that each time this constraint is activated, the sequence of generated numbers becomes less random because of these repeating two numbers.

9.1 Log entropy measurements

In [25] authors discuss the use of entropy and entropy rate as a measure of log variability. The tool presented in their paper allows to compute different metrics from logs. Unlike our metrics, it doesn’t support data, but can process larger logs, and results are independent from model meaning that we can compare entropy of logs obtained from different models.

Using this tool, we will compare logs by our generators for the business trip model with 3 real logs: NASA Crew Exploration Vehicle, Sepsis Cases and BPI2017 Challenge. Entropy means degree of randomness. Therefore, higher values table mean better result. The results are shown at the table 8.

	NuSMV	Alloy with shuffling	Alloy	NASA CEV	Sepsis Cases	BPI2017
Log Entropy(H)-(trace-based)	8.4896	8.2239	4.9453	11.265	9.334	11.9931
Log Entropy(H)-(prefix-based)	10.7348	9.5445	6.6745	10.101	10.2271	12.934
Log Entropy(H)-(all block-based)	13.2594	10.8403	8.3035	14.7233	14.5012	16.4566
Log Entropy(H)-(Nearest Neighbor: Kozachenko-Leonenko)	44.7697	31.5661	30.6619	23.2223	23.7862	33.2025
Log Entropy Rate(h)-(k-block: difference-based using cutoff 1)	1.7894	1.456	0.8013	1.5563	1.8369	1.3031
Log Entropy Rate(h)-(k-block: ratio-based using cutoff 1)	2.7393	2.5523	2.6773	5.0989	3.2382	3.7863
Log Entropy Rate(h)-(k-block: ratio-based using cutoff 5)	1.2432	1.015	0.7172	1.145	0.4854	0.6042

Table 8 – log entropy and entropy rate measured by tool from [25]

As we can see, general trend is following: entropy of our synthetic logs and real ones is comparable. NuSMV has slightly better results. Alloy without constraint shuffling performs worse.

10 Execution time evaluation

In this section, we will compare the time performance of the generation tools presented in [1], [5] and [6], and our generators.

In the first step we present the models used in the evaluation and encode them in our input format. Then we compare the time performance of the proposed generators with state-of-the-art approaches. In detail we first compare the Alloy generator with respect to the state-of-the-art generators on the same models used in the papers. We then provide a preliminary evaluation of the NuSMV generator by comparing it with the performance of the Alloy generator on similar parameters. Finally, we investigate the impact of model parameters on the performance of the generators

10.1 Models for execution time measurements

As different generators support different set of constraints, features, and input formats, it is not possible to have a universal model to run on all generators. Therefore, we have 5 different models for comparing our generators with others.

Model M1. Business trip model. The original model presented in [1,2] uses the DPIL language. We rewrote in Declare with exactly the same semantics. The model has 6 activities, 3 data attributes (enumerative only), 7 data bindings and 15 constraints. 7 of these constraints have data.

Model M2. Fracture treatment model. The original model is presented in [5] and also use Declare language (but without data), so we translated it in our format. The model has 8 activities and 8 constraints.

Model M3. Acme travel company model. This model was used in [6]. Since this model is originally an Alloy model, it is not possible to fully represent it in Declare, so we slightly modified the intermediate .als file to have an exact match for testing purposes. This model has 11 activities, 30 constraints and no data.

Model M4. Business trip model (modified). We modified the business trip model from [1, 2] by adding more activities, different constraints and data. It has 8 activities, 4 data attributes and 17 constraints. 2 of these constraints use data.

Model M5. Loan application model. This model has 5 activities, 7 data attributes and 15 constraints, 4 of them also constrain data.

All the models can be found in the repository https://github.com/dark-soullock/MPDeclareLogGenerator/tree/master/data

10.2 Comparison with the state of the art

In this section we compare our Alloy log generator with the tool presented in [1],[5]. The tool presented in [1] has as input parameters trace length (L) and number of traces (N). L represents the maximum length of the generated traces. It uses business trip model M1 described above (cannot be changed), maximum traces length as 10..80 and amount of traces 10..1000. Therefore, for comparison we will use the same model and generation parameters.

In practice, given L, the tool produces traces of length lower than L (average length given in the column ‘Actual L’ at Figure 14), with maximum length of 2^(Round(Log[2,L])-1)-2. The actual length of generated traces is present in column ‘L range’ of table on Figure 14.

The minimum length (6) bounded only by constraints of the model (i.e. no traces with less than 6 activities exist for the model).

We will have two measurements in our tool: one with the same input parameters (trace length from 2 to max), and another one with different parameters, where minimum and maximum trace length set according to actual minimal and maximal values (column 'L range' at Figure 14) generated by tool presented in [1] so as to generate a comparable output (column 'our result with altered parameters').

Input parameters		Schoenig's result			Our result			ur result with altered parameter		
L	N	Actual L	L range	Time (s)	Actual L	Time (s)	Time %	Actual L	Time (s)	Time %
10	10	6	6..6	1.8	9.5	2.5	138.89%	6	1.4	77.78%
20	10	10.7	6..14	19	19	3.3	17.37%	14	2.7	14.21%
20	100	13.33	6..14	24	19.5	5.3	22.08%	13.6	4.1	17.08%
20	1000	13.9	6..14	55	19.7	13.9	25.27%	13.9	10.6	19.27%
30	10	9.3	6..14	68	27.7	4.6	6.76%	14	2.7	3.97%
30	100	13	6..14	74	29.8	6.4	8.65%	13.6	4.1	5.54%
30	1000	13.9	6..14	118	29.6	17.3	14.66%	13.9	10.6	8.98%
40	10	22	7..30	153	38	5	3.27%	29.4	3.9	2.55%
40	100	28.45	7..30	173	39.4	7.6	4.39%	29.8	6.8	3.93%
40	1000	29.8	7..30	284	39.6	22.3	7.85%	29.9	18.1	6.37%
50	10	20.8	10..30	348	46.7	6.8	1.95%	29.2	4	1.15%
50	100	28	10..30	374	49.3	10	2.67%	29.8	6.4	1.71%
50	1000	29.8	10..30	535	49.6	28.4	5.31%	29.9	17.2	3.21%
60	10	26.5	9..30	604	56	8.8	1.46%	29.9	4.2	0.70%
60	100	29.6	9..30	627	58.3	11.8	1.88%	29.8	6.4	1.02%
60	1000	29.6	9..30	850	59.7	33.8	3.98%	29.9	17.7	2.08%
70	10	50	32..62	1148	63.3	11	0.96%	59.2	9.4	0.82%
70	100	59.2	32..62	1251	68.1	14.9	1.19%	61.9	12.4	0.99%
70	1000	61.7	32..62	1664	69.6	41.7	2.51%	61.9	35	2.10%
80	10	45.8	30..62	2005	68	14.7	0.73%	61.6	8.4	0.42%
80	100	57.4	30..62	2159	77.5	18.7	0.87%	61.6	11.9	0.55%
80	1000	61.5	30..62	2687	79.5	52.1	1.94%	61.9	36.7	1.37%

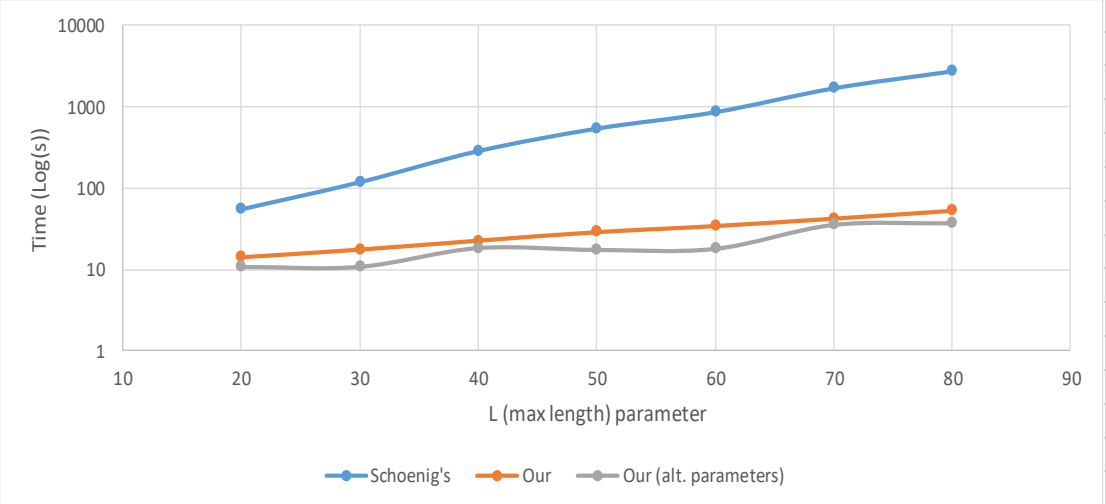


Figure 14 – Alloy generation time comparison with baseline

Figure 14 shows the execution time (in seconds) required for generating logs of 1000 traces from model M1, each one containing traces of a given length (from 10 to 80). Unfortunately, the tool in [1] managed to produce only 4 traces with maximum length 10, therefore we start

reporting values from 20 on the plot. Note, that time axis has a logarithmic scale. The Figure shows that our Alloy generator always performs better than the tool in [1].

Table 8 shows execution times in milliseconds for generating log from model M2 with the tool presented in [5] and our Alloy-based tool. Note that in this experiment we do not include measurements for the generation of 10 traces because measuring error for such short periods is too high.

Max. length	# traces	Time, ms. [5]	Time, ms. (our)
10	100	15	2747
10	1000	123	6260
20	100	21	4915
20	1000	178	11753
30	100	38	6168
30	1000	332	13729
40	100	49	7582
40	1000	772	17294
50	100	55	10058
50	1000	585	19818
60	100	69	10608
60	1000	589	22474
70	100	79	14470
70	1000	569	29385
80	100	83	18891
80	1000	653	36651

Table 9 – Comparison of Alloy and FSA log generation time

[5] always performs better, requiring less time than the Alloy generator. However, [5] only supports standard Declare and does not support MP-Declare. Another disadvantage of the too in [5] is that vacuity cannot be disabled.

The approach presented in [6] is also based on Alloy, but it does not support data. In addition, it is designed to generate only one trace and it does not have a tool that can be used outside of Alloy (trace can be viewed in the Alloy analyser interface and cannot be saved in a file). To compare this approach with ours, we used the model they use (model M3 – "Acme Travel Company" case from [19]). We generate Alloy code in our tool, and then run both models in the Alloy analyser. As generating one trace takes a very low amount of time, we calculate the average of 10 runs.

Alloy gives us two timespans: time for the generation of CNF, and time for solving it. the CNF is generated only once for log, while the solving step occurs for each trace. All timings are in milliseconds.

Their solution		Our solution	
Gen. CNF	Solving	Gen. CNF	Solving
503.1	150	684.4	107.9

Table 10 – Comparison of generation time of one trace in our and [6] solution

Table 10 shows the execution times for generating the CNF and solving it (to obtain 1 trace) with the two approaches. We can observe that the results are comparable.

10.3 Comparison with NuSMV

As NuSMV does not support the generation of multiple counterexamples, we have made a preliminary performance evaluation by running the generation of the same trace N times. The generation of the same amount of different traces cannot take less time because for each new trace we will need to negate all the previous ones, which will cause a model growth.

To perform the preliminary performance test, we used the business trip model (model M4). In this test we didn't add inverted traces to the model, so all the traces in the log generated by NuSMV are the same.

Table 11 shows execution times (in seconds) in comparison with Alloy for maximum trace length equal to 40.

#traces	Time s. NuSMV	Time s. Alloy
10	33	4.7
100	326	8
1000	3260	23

Table 11 – Generation time of N traces in NuSMV (no trace negation) and Alloy for model M4

NuSMV is slower, but it allows to use constraints on numbers, whose support is limited in Alloy.

Table 11 shows the execution time in milliseconds of NuSMV with trace negation in comparison with Alloy. Due to the long generation time of NuSMV we did not perform the test for 1000 traces and with trace length greater than 40.

Max. length	# traces	Time, ms. NuSMV	Time, ms. Alloy
10	10	10998	3107
10	100	138052	4048
20	10	38229	4256
20	100	714432	6605
30	10	128209	5683
30	100	3089107	7076
40	10	346852	6945
40	100	9515276	9606

Table 12 – Generation time of N traces in NuSMV and Alloy from MP-Declare model

As we can see in Table 12, generation time of NuSMV grew quadratically with the amount of traces, which makes it less suitable for generating large logs.

10.4 Impact of model parameters on the execution times

The main elements in an MP-Declare model are activities and constraints. As our generator supports data, we also have data attributes and data constraints. In order to determine how they influence the performance, we use a set of models with different characteristics. In order to improve the precision of the measurements, we performed each generation 4 times. Finally, we apply linear regression (with least squared error cost function) to fit the data, and look at the coefficients.

Varying the number of activities. For this test the initial model contains 10 activities. In the N-th test we add N activities and an existence constraint on them, to make sure that they will appear in the log. Figure 15 shows the generation time in milliseconds for N=1..10.

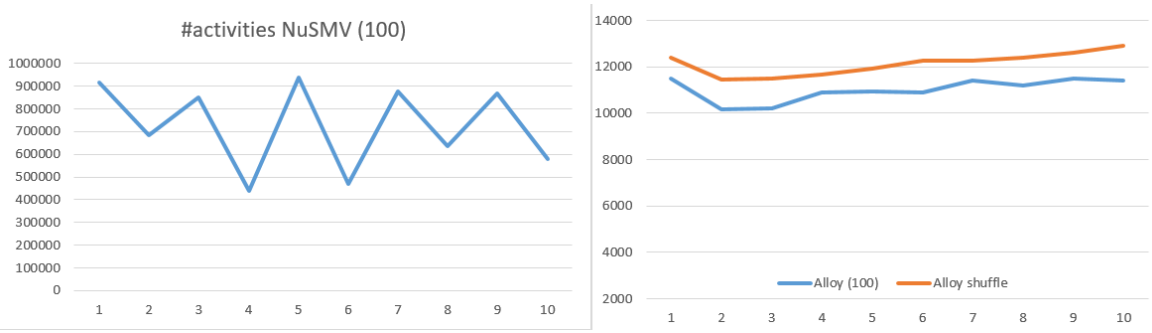


Figure 15 – generation time for N activities

As we can see, for NuSMV trend is slightly negative, and for Alloy is positive, but overall the lines are almost flat. We can observe, that NuSMV generation time is significantly less with even amount of activities. From regression coefficients, it takes 0.228 ms/trace more time for each added activity for Alloy, and -29.8 ms/trace less for NuSMV.

Varying the number of constraints. For this test the initial model contains 11 activities: a1..a10, b and one constraint Existence[b] (only for b, to ensure activation). In the N-th model we add N response constraints as Response[b, a1], Response[b, a2], … Response[b, aN]

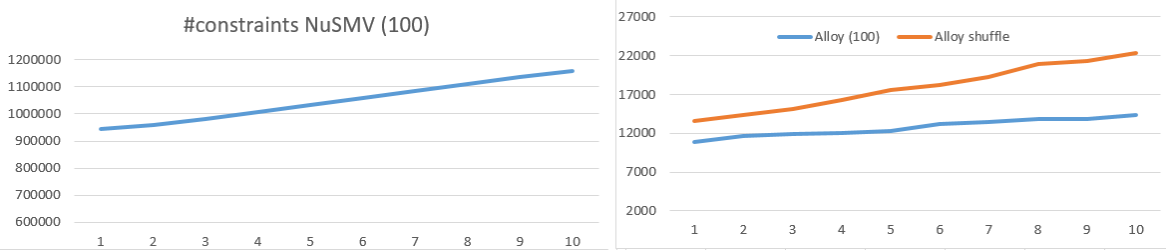


Figure 16 – Generation time for N constraints

Figure 16 shows the result. We can see that both – SMV and Alloy perform worse with more constraints. For SMV the time increases by 61.86 ms/trace for each added constraint, and for Alloy by 0.939 ms/trace.

Adding data to constraints. The initial model for this test is similar to the previous one, but includes ten enumerative data attributes, each of them with two possible values and bound to one activity. It also contains ten response constraints. In the N-th model we add data to the N-th constraint. So Response[b, aN] is transformed to ‘Response[b, aN X]||X.xN is xN1’, which means that activity ‘b’ should be followed by activity ‘aN’ (where N is number) with data attribute xN limited by one of two values. I.e. if in activity A data attribute X has possible values Y and Z, then in our constraint we specify that after activity b, A must occur with X equal to Y (but not Z).

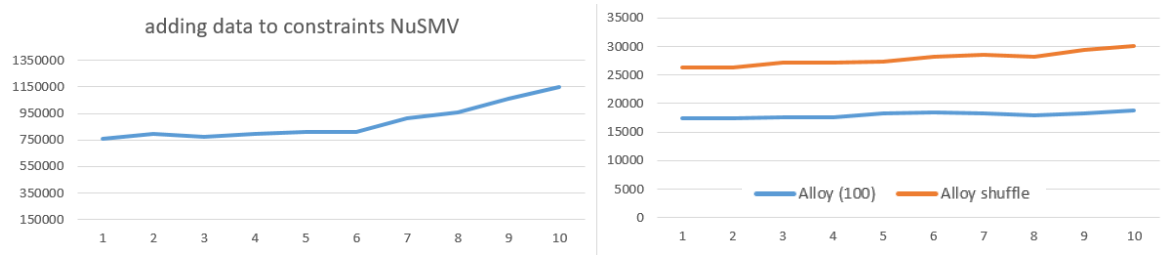


Figure 17 – Generation time for N functions in data constraints

We can see a growth for SMV (100.99 ms/trace) and almost a flat line for Alloy (0.305 ms/trace)

Varying enumerative data constraints. The initial model for this test is similar to the original model of the previous test, but does not include constraints. For the N-th model we add 'Response[b, aN X]||X.xN is xN1'

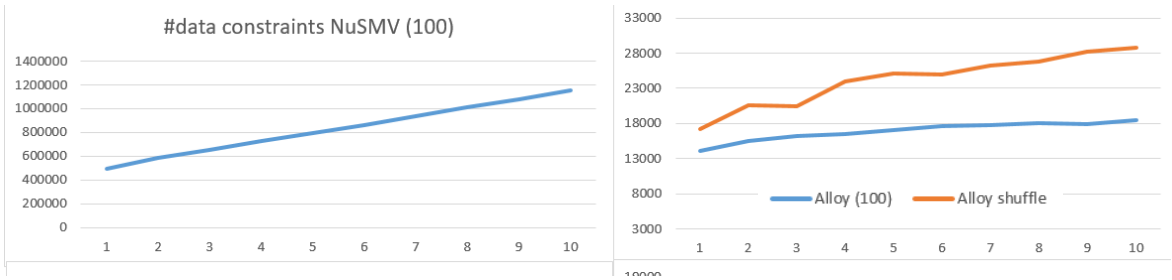


Figure 18 – Generation time for N data constraints

Figure 18 shows the results. As we can see, the time increases when the amount of data constraints increases. The Figure looks similar to the one in the previous experiment, and shows a steady growth. However, the growth rate is greater than for constraints without data, and equal to 1.058 ms/trace for Alloy, and 179.69 ms/trace for NuSMV.

Varying numeric data constraints. The initial model for this test is the same as in the previous test, but all data attributes are numeric (of type integer) instead of enumerative. In the constraints we limit the values to the positive numbers.

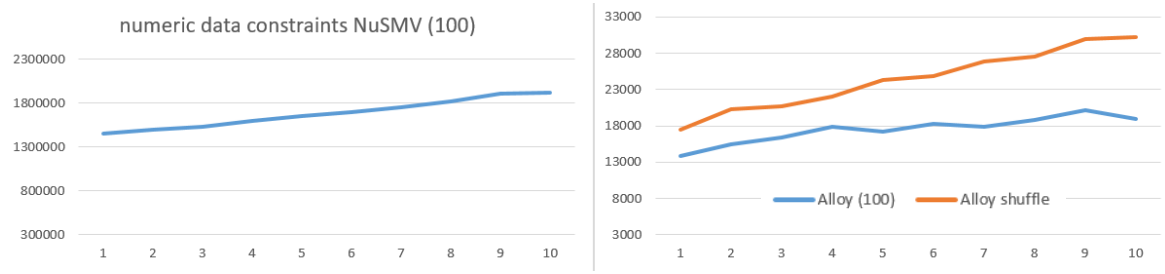


Figure 19 – Generation time for N numeric data constraints

The results are again similar to previous tests, but for Alloy the growth rate is bigger (1.4 ms/trace), and for NuSMV is smaller (136.43 ms/trace) for numeric data constraints than one for enumerative. Increase in time for Alloy can be explained by the time needed for pre/post-processing steps, and by the fact that there are some workarounds in the model checker which allow us to arrange numeric constraints.

Varying unconstrained data attributes. The initial model for this test contains 10 activities with existence constraints. In the N-th model we add 2 data attributes, each of them bound to the N-th activity. The type of attribute does not matter as we do not have any data constraint.

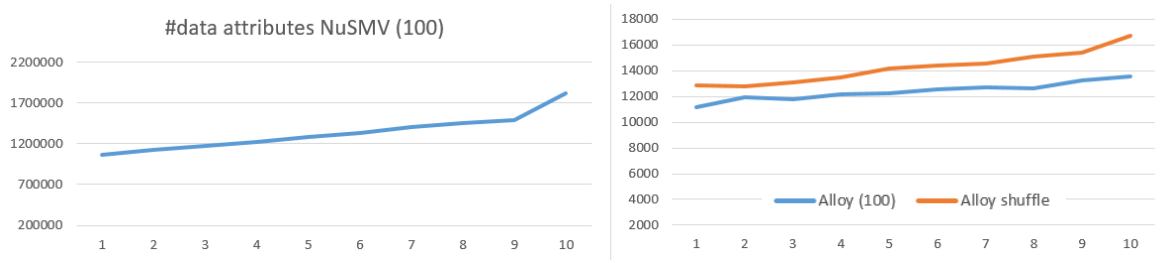


Figure 20 – Generation time for N data attributes

Figure 20 shows the results. In this plot we see a growth again. In this case, the rate is 0.554 ms/trace for Alloy, and 171.96 for SMV.

Varying the number of values in enumerative data attributes. For this test we have ten activities a1..a10 with existence constraints on them. Each activity has two data attributes – x1 and x2. For each next model we add one value to these data attributes, so in the N-th model x1 and x2 have N possible values.

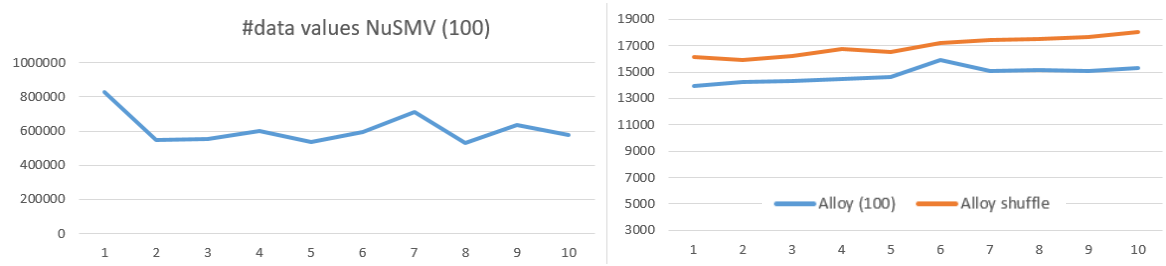


Figure 21 – Generation time for N values presented in data attributes

Figure 21 shows the results. In NuSMV the time is slowly decreasing with more values by -20.63 ms/trace, and for Alloy it is increasing by 0.379 ms/trace. The way enumerative data attributes are encoded in our NuSMV based generator is exactly the same as the activities are encoded, so this could be expected.

Varying the constraint types. In this test we measure the time for the generation of logs for constraints of different types. The initial model consists of eleven activities, and an existence constraint to ensure the activation. For each test we add one constraint, a different type for each test.

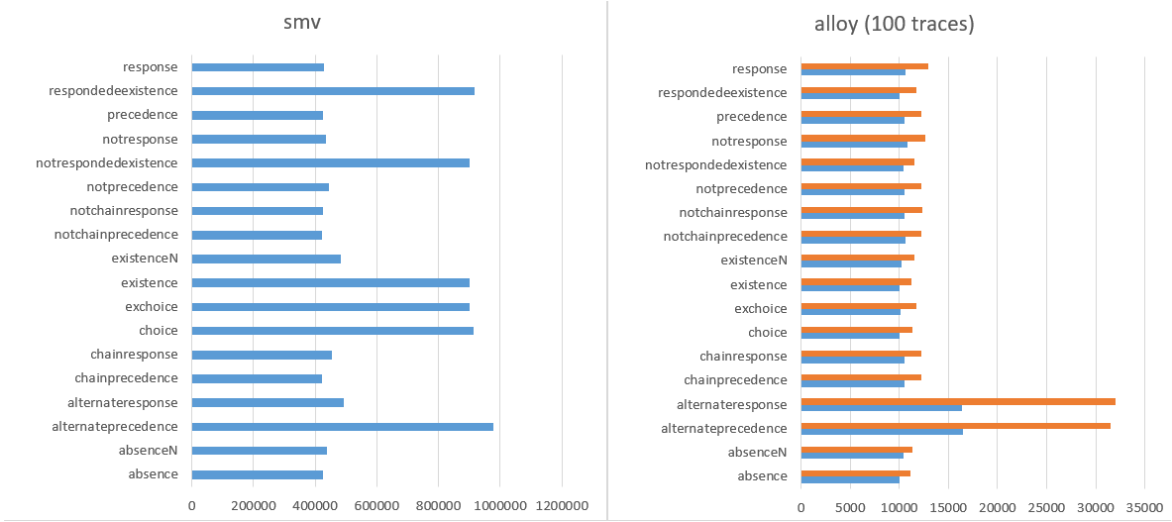


Figure 23 – Impact of different constraint types on generation time

Figure 23 shows the results. On the X axis we have generation time. As we can see, for NuSMV, the constraints that require more time are: responded existence, not responded existence, existence, exclusive choice, choice and alternate precedence. For Alloy less efficient constraints are alternate response and alternate precedence.

Summing up. Now with all these measurements we can summarize the impact of the different kind of model characteristics on generation time.

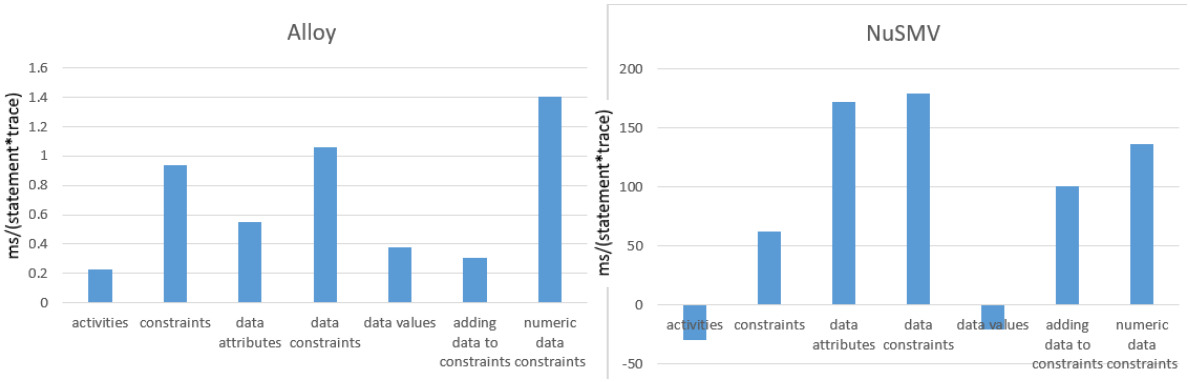


Figure 22 – Impact of statements in the model on generation time

Activities and data attribute values affect the performance least. For NuSMV data attributes and data constraints have the highest influence on performance. Biger impact of data attributes can be explained with the fact, that in NuSMV all data attributes are present in all activities. The biggest impact of Alloy performance is given by numeric data constraints.

Overall, Alloy generator performs better in most cases. But as we use intervals for encoding numbers, with a lot of numeric data constraints we will get unevenly distributed numbers. NuSMV generates the same amount of traces much longer, but produces slightly better logs (in terms of variability) for models without data, and allows to use operations on numbers if we want dependent data attributes (i.e. data attributes ‘price’ and ‘discounted price’). Disadvantage of NuSMV generator is that it does not allow to express correlation between two data attributes in different activities (I.e. A should be followed by B with the same value of attribute X).

11 Conclusions

In this paper we reviewed different approaches for the declarative log generation, existing in other papers. From these works we chose the most suitable parts, added more generic data support, and improved performance. After this, the found solution based on the Alloy analyzer was wrapped in a java application. This application supports input of declarative models in a textual language specifically designed and proposed in this thesis for dealing with multiperspective declarative constraints, does all necessary pre and post processing, and saves the results as a .xes file. Besides the Alloy-based solution we also developed a generator based on the NuSMV model checker. Finally, performance and quality log evaluation was carried out to understand and collect the main characteristics of the generators and to compare our solutions with each other and with existing state of the art techniques.

For future work I would like to approach the problem from the side of first order logic languages like Prolog, or to use SMT model checkers, which can handle numbers better. Also, it might be possible to improve performance of our tool by using the approach used in [6] for encoding trace (using standard implementation of ordered sequences in Alloy for storing traces).

12 References

[1] Ackermann L., Schönig S., Jablonski S. (2017) Simulation of Multi-perspective Declarative Process Models. In: Dumas M., Fantinato M. (eds) Business Process Management Workshops. BPM 2016. Lecture Notes in Business Information Processing, vol 281. Springer, Cham

[2] Ackermann L., Schönig S., Jablonski S. (2017) Towards Simulation- and Mining-based Translation of Resource-aware Process Models. In: Dumas M., Fantinato M. (eds) Business Process Management Workshops. BPM 2016. Lecture Notes in Business Information Processing, vol 281. Springer, Cham

[3] Ackermann L., Schönig S., Jablonski S. (2016) Towards Simulation- and Mining-Based Translation of Process Models. In: Pergl R., Molhanec M., Babkin E., Fosso Wamba S. (eds) Enterprise and Organizational Modeling and Simulation. EOMAS 2016. Lecture Notes in Business Information Processing, vol 272. Springer, Cham

[4] Andrea Burattin,Fabrizio M. Maggi,Alessandro Sperduti (2016) Conformance checking based on multi-perspective declarative process models In: Expert Systems with Applications Volume 65, Pages 194-211

[5] C. Di Ciccio, M. L. Bernardi, M. Cimitile, and F. M. Maggi, “Generating event logs through the simulation of declare models,” in EOMAS, pp. 20–36, 2015.

[6] Yoann Laurent, Reda Bendraou, Souheib Baarir, Marie-Pierre Gervais. Planning for Declarative Processes. SAC’14 - The 29th Annual ACM Symposium on Applied Computing, Mar 2014, Gyeongju, South Korea. ACM, pp.1126-1133, 2014,<10.1145/2554850.2554998>.<hal-01088183>

[7] Ubaier Ahmad Bhat (2016) Runtime Monitoring of Data-Aware business rules with Integer Linear Programming

[8] Federico Chesani, Anna Ciampolini, Daniela Loreti, Paola Mello, "Abduction for generating synthetic traces"

[9] Andrea Burattin. "PLG2: Multiperspective Process Randomization with Online and Offline Simulations". In Online Proceedings of the BPM Demo Track 2016; Rio de Janeiro, Brasil; September 18, 2016; CEUR-WS.org 2016.

[10] Andrea Burattin. "PLG2: Multiperspective Processes Randomization and Simulation for Online and Offline Settings". In CoRR abs/1506.08415, Jun. 2015.

[11] Andrea Burattin and Alessandro Sperduti. "PLG: a Framework for the Generation of Business Process Models and their Execution Logs". In Proceedings of the 6th International Workshop on Business Process Intelligence (BPI 2010); Stevens Institute of Technology; Hoboken, New Jersey, USA; September 13, 2010. 10.1007/978-3-642-20511-8_20.

[12] Westergaard, M., Slaats, T.: Cpn tools 4: A process modeling tool combining declarative and imperative paradigms. In: BPM (Demos). (2013)

[13] Song M., Günther C.W., van der Aalst W.M.P. (2009) Trace Clustering in Process Mining. In: Ardagna D., Mecella M., Yang J. (eds) Business Process Management Workshops. BPM 2008. Lecture Notes in Business Information Processing, vol 17. Springer, Berlin, Heidelberg

[14] Bose R.P.J.C., van der Aalst W.M.P. (2010) Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models. In: Rinderle-Ma S., Sadiq S., Leymann F. (eds) Business Process Management Workshops. BPM 2009. Lecture Notes in Business Information Processing, vol 43. Springer, Berlin, Heidelberg

[15] R.P., Jagadeesh Chandra Bose & Aalst, Wil M. P.. (2009). Context Aware Trace Clustering: Towards Improving Process Mining Results. SDM. 10.1137/1.9781611972795.35.

[16] D. Jackson. Software Abstractions: logic, language and analysis. Mit Pr, 2011.

[17] N. E´en and N. S¨orensson. An extensible sat-solver. In Theory and applications of satisfiability testing, pages 502–518. Springer, 2004.

[18] 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

[19] W. M. Van Der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. Springer, 2006.

[20] D. Jackson, Software Abstractions: logic, language, and analysis. MIT press, 2012.

[21] U. Frank, "Multi-perspective enterprise modeling (memo) conceptual framework and modeling languages," in HICSS, pp. 1258–1267, 2002.

[22] S. Scho¨nig, C. Cabanillas, S. Jablonski, and J. Mendling, "Mining the organisational perspective in agile business processes," in BPMDS, pp. 37–52, 2015.

[23] Awad A., Smirnov S., Weske M. (2009) Resolution of Compliance Violation in Business Process Models: A Planning-Based Approach. In: Meersman R., Dillon T., Herrero P. (eds) On the Move to Meaningful Internet Systems: OTM 2009. OTM 2009. Lecture Notes in Computer Science, vol 5870. Springer, Berlin, Heidelberg

[24] Rényi, A. Acta Mathematica Academiae Scientiarum Hungaricae (1953) 4: 191. https://doi.org/10.1007/BF02127580

[25] Back C.O., Debois S., Slaats T. (2018) Towards an Entropy-Based Analysis of Log Variability. In: Teniente E., Weidlich M. (eds) Business Process Management Workshops. BPM 2017. Lecture Notes in Business Information Processing, vol 308. Springer, Cham

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Vasyl Skydanienko**,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Data-aware Synthetic Log Generation for Declarative Process Models,

(title of thesis)

supervised by Fabrizio Maria Maggi, Chiara Di Francescomarino, Chiara Ghidini

(supervisor's name)

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

23.05.2018