Declare MoGeS: Model Generator and Specializer

Manal Laghmouch\textsuperscript{1,2,*}, Benoît Depaire\textsuperscript{1}, Nicola Gigante\textsuperscript{3}, Mieke Jans\textsuperscript{1,2} and Marco Montali\textsuperscript{3}

\textsuperscript{1}Hasselt University, Martelarenlaan 42, 3500 Hasselt, Belgium
\textsuperscript{2}Maastricht University, Minderbroedersberg 4-6, 6211 LK Maastricht, Netherlands
\textsuperscript{3}Free University of Bozen-Bolzano, Piazza Università, 1, 39100 Bolzano BZ, Italy

Abstract
This demo introduces Declare MoGeS, an automated approach for generating and specializing Declare process models that can be employed as input for log generation. The specialization of Declare models is particularly interesting to produce event logs that encompass a subset of the behavior of other logs. Declare MoGeS seamlessly integrates with existing log generators, streamlining the log generation process.

Keywords
Declare, Model Generation, Model Specialization, Linear Temporal Logic

1. Introduction

Comparative evaluations have become increasingly important in assessing the strengths and weaknesses of process discovery algorithms (cf. [1]). Synthetic event logs are important for advancing research, with their primary utility lying in the creation of logs that exhibit specific characteristics, facilitating the evaluation of process discovery algorithms.

To generate synthetic logs, process models are required as input. In a declarative context, this translates to the availability of declarative process models [2]. Currently, log generators are fed with manually designed process models. If one wants to create event logs with different characteristics, different input models are required. Manually creating or adapting process models is a time-consuming, and consequently, costly task.

This demo proposes Declare MoGeS, a first-of-its-kind tool that enables the automatic generation and specialization of declarative process models. The first objective of the demo is to generate artificial models while having control over the models’ main characteristics. Furthermore, to ensure control over subsets of process behaviors, it becomes essential that these process models can be crafted in a manner where some models are considered as specializations of others. Specialization refers to restricting the allowable behavior of a process model. Model A which is a specialization of model B, allows for less behavior than model B. Presently, model
specialization primarily involves adding constraints to an initial model, resulting in limited variations of process models. To address this limitation, the second objective of the demo is to propose an automated approach to generate specializations by adapting constraints from an initial model, thereby enabling controlled variations [3, 4]. For instance, a constraint stating that activity a should be followed by activity b (i.e. Response(a,b)) can be specialized by requiring immediate occurrence of activity b after a (i.e. ChainResponse(a,b)).

2. Innovations and Main Features

Given that the demo has to be able to (1) generate artificial declarative process models and (2) specialize declarative process models that can serve as input to generate event logs, the developed DECLARE Model Generator and Specializer (Declare MoGeS) adheres to the following requirements.

- **Declarative Modeling Language** – describe business processes in a flexible declarative language (DECLARE).

- **Consistency** – the generated and specialized models only consist of non-contradictory constraints.

- **Specialization of Process Models** – enable refinement and tailoring of model behavior.

- **Balance Between Randomness and User Control** – allow for variations of process models, and, at the same time, enough control over the generated models.

- **Compatibility Output Models with Existing Log Generators** – The output model is a DECLARE model saved in a file format1 suitable as input for existing log generators.

In the following subsections, we describe the algorithms behind Declare MoGeS.

2.1. Generating a Random Declare Model

Algorithm 1 shows that a desired number of activities and constraints (*alphabet_size* and *set_size*), a set of DECLARE templates that can be selected to generate a model (*template_list*), and the probability that a particular DECLARE template is chosen (*initial_prob*) serve as inputs for model generation.

Model Generation starts with initializing an empty list of DECLARE constraints. This list will eventually form the created model. Next, a DECLARE constraint is selected by randomly choosing a template from *template_list*, taking into account the *initial_prob*. Afterward, activities from an alphabet of size *alphabet_size* are chosen to obtain a new *constraint*.

The new *constraint* is added to the model IF it complies with the following two key conditions. First, the new *constraint* must be *consistent* with the constraints already present in the model, i.e., their conjunction must be *satisfiable*. We refer to the existing set of constraints as the temporary model. For instance, consider a temporary model consisting of the constraints

---

1.decl file format
Input: Size of the alphabet of activities: alphabet_size
    Number of DECLARE constraints: set_size
    List of DECLARE constraint templates: template_list
    Initial probability of choosing templates: initial_prob
    Number of subsequent tries to add a constraint: x

Output: Set of DECLARE constraints: constraints

Initialize:
    constraints = []
    j ← 0
    n ← 0
    while j < set_size do
        new_constraint = random(template_list, initial_prob, activities)
        if new_constraint is consistent w.r.t. constraints and
          new_constraint is not redundant w.r.t. constraints then
            constraints ← constraints ∪ new_constraint
            j ← j + 1
            n ← 0
        else
            n ← n + 1
            if n > x then
                print No model found with the given parameters
                return constraints
    return constraints

Algorithm 1: Model Generator

[Response(a,b), ChainResponse(b,c)]. The constraint ChainResponse(b,d) would be inconsistent because it contradicts ChainResponse(b,c). Second, the new constraint should not be redundant. For example, ChainResponse(b,d) (i.e. if b occurs, then d should occur in the next position) implies Response(b,d) (i.e. if b occurs, then d should occur eventually after b). In this case, adding Response to the model when a ChainResponse is already included is redundant.

Both conditions, i.e. consistency and non-redundancy, are checked with BLACK [5] by using the Linear Temporal Logic over finite traces (LTLf) encoding of the DECLARE constraints. If both conditions are met, the new_constraint is added to the model, or discarded otherwise.

The algorithm keeps track of how many subsequent times a constraint is discarded (n). This process continues until the set_size is met (model is returned) or until x times in a row, a new_constraint cannot be added to the model (a message is shown to the user and the model [constraints] is returned).

2.2. Specializing a Declare Model

Algorithm 2 shows the process for specializing a DECLARE process model. To specialize a model, the user provides an initial model consisting of constraints that need to be specialized
(constraints). Optionally, the user can input a set of constraints from the initial model that should be kept in the specialized model (model). Furthermore, a specialization percentage (specialization_percent) that defines the probability a constraint will be specialized is set.

**Input:** Initial `declare` model: `constraints`  
Specialization percentage: `specialization_percent`  
Initial specialized model: `model`  

**Output:** A specialization of `constraints`  
for each `initial_constraint` in `constraints` do  
  if `initial_constraint` can be specialized then  
    if `random() < specialization_percent` then  
      Generate specialized  
      if `specialized` \( \notin \) `model` then  
        `model` \( \leftarrow \) `model` \( \cup \) `specialized`  
      else  
        `model` \( \leftarrow \) `model` \( \cup \) `initial_constraint`  
    else  
      if `initial_constraint` \( \notin \) `model` then  
        `model` \( \leftarrow \) `model` \( \cup \) `initial_constraint`  
  else  
    if `initial_constraint` \( \notin \) `model` then  
      `model` \( \leftarrow \) `model` \( \cup \) `initial_constraint`  

return `model`

Algorithm 2: Model Specializer

The process of specialization (algorithm 2) starts with an `initial_constraint` from `constraints`. If `initial_constraint` can be specialized, then a specialization is added to the `model` in some cases. The `specialization_percent` is taken into account to determine whether a specialization should be added or not. Otherwise, the `initial_constraint` is added to the `model`. This process ends when all constraints from the initial model are considered. The specialized model `model` is a specialization of the initial model `constraints`.

3. Maturity

Declare MoGeS is implemented in Python and stored in a GitHub repository\(^2\). Additionally, a comprehensive video tutorial demonstrating the tool’s usage can be found within the same repository, providing users with an informative resource for getting started with Declare MoGeS.

In computational tests, we tested the Declare MoGeS by conducting a total of 2392 runs, each aimed at artificially generating and automatically specializing each of the generated `DECLARE` process models at four distinct percentages (30%, 50%, 70%, and 100%). Approximately 75% of the runs resulted in the generation of models containing between 5 to 25 constraints, all achieved within an 11-minute time frame. Furthermore, it’s worth noting that models with fewer than 16 constraints were generated almost instantly, with a median time of less than a second. However,

\(^2\)https://github.com/manallaghmouch/DeclareMoGeS
for models comprising more than 35 constraints, the execution time could exceed an hour. This prolonged execution was primarily attributed to the computationally intensive consistency and non-redundancy checks performed by BLACK.

On the other hand, the Model Specializer displayed efficiency throughout our tests, consistently boasting running times of less than one second for all specialization percentages. These results highlight the effectiveness of specialization through adapting constraints.

4. Conclusion and Future Work

This paper presents a novel approach for automatically generating and specializing DECLARE process models to facilitate log generation. The effectiveness of the approach is demonstrated and evaluated, highlighting its ability to swiftly generate and specialize DECLARE models containing 5 to 25 constraints.

In future research, there are opportunities to expand. One potential avenue involves incorporating a data-aware aspect. After integration, studies can evaluate data-aware process discovery algorithms using logs generated from data-aware input models. Additionally, it is interesting to extend the study beyond the predefined templates offered by the DECLARE language. Future research will delve into exploring LTL formulas that surpass the existing templates.

Acknowledgments

Manal Laghmouch thanks Research Foundation - Flanders for the SB PhD fellowship (IS40622N) granted to support this research. Nicola Gigante acknowledges the support of the PURPLE project, 1st Open Call for Innovators of the AIPlan4EU H2020 project, a project funded by EU Horizon 2020 research and innovation programme under GA n. 101016442 (since 2021).

References