

Two Solutions for Checking LTL_f Properties in Event Logs

Tedi Ibershimi^{1,*}, Diellsimeone Xhemalaj¹, Anti Alman², Ivan Donadello¹ and Fabrizio Maria Maggi¹

¹Free University of Bozen-Bolzano, Bolzano, Italy

²University of Tartu, Tartu, Estonia

Abstract

Conformance checking is an important task in process mining that checks whether a case of a business process is compliant or not with a process model. Process models in conformance checking can be expressed by using the procedural paradigm (e.g., through BPMN models or Petri nets) or the declarative one (e.g., through DECLARE models). Declarative process modeling languages represent the process behavior using temporal logic constraints mainly expressed with Linear Temporal Logic on finite traces (LTL_f). In this paper, we present two solutions for checking LTL_f temporal properties over the traces in an event log. The two solutions have been implemented in the RUM Java toolkit (equipped with a Graphical User Interface) and in the DECLARE4PY Python library. We also preliminary evaluate the time performance of both implementations showing that the execution times are reasonable for sufficiently complex checking tasks.

Keywords

Linear Temporal Logic, Conformance Checking, Log Filtering

1. Introduction

Process mining [1] focuses on the analysis of business processes based on event logs that contain information about process executions. An important component in process mining is a process model. Traditional process models follow the procedural paradigm and are suitable for representing predictable and stable business processes through an explicit specification of the allowed behaviors. The procedural paradigm is well suited for business processes where all possible executions are relatively similar to each other. A shortcoming of the procedural paradigm is the difficulty of representing processes that have high variability among all possible executions as each variation needs to be explicitly accounted for, thus leading to high complexity in the process models [2]. In contrast, declarative process models can be more easily used to manage this type of processes since they are expressed through a set of constraints that state what the process behavior must not contradict, thus enabling the compact representation of multiple allowed process executions.

ICPM Doctoral Consortium and Demo Track 2023, October 23-27, Rome, Italy


*Corresponding author.

✉ tibershimi@unibz.it (T. Ibershimi); dxhemalaj@unibz.it (D. Xhemalaj); anti.alman@ut.ee (A. Alman); ivan.donadello@unibz.it (I. Donadello); maggi@inf.unibz.it (F. M. Maggi)

ORCID 0000-0002-5647-6249 (A. Alman); 0000-0002-0701-5729 (I. Donadello); 0000-0002-9089-6896 (F. M. Maggi)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

An important task in process mining is conformance checking, whose goal is to find discrepancies between a process model and the actual executed process as recorded in an *event log*. In this paper, we consider the verification of an LTL_f model M (i.e., a set of LTL_f constraints) with respect to a log L containing traces each being a sequence of events (activity executions). LTL_f checking verifies whether each trace in L satisfies all constraints in M .

The presented LTL_f checking tools provide a list of patterns that have been shown to be particularly useful in the literature [3, 4]. In particular, each pattern is represented as an LTL_f template, i.e., a formula with placeholders to be substituted by concrete activities to obtain a specific constraint. The family of simple LTL_f patterns (hereafter called $SLTL_f$) were first provided in [3], whereas the DECLARE patterns were first introduced in [4]. The checkers also support Branched DECLARE (hereafter called B-DECLARE) in which DECLARE templates are instantiated with disjunctions of activities $a = \bigvee_{i=1}^n a_i$.

The LTL_f checkers have been implemented in the RUM Java toolkit [5] and the DECLARE4PY Python library [6]. These tools can be used for checking LTL_f formulas in the traces of an event log, but can also be used as advanced log filters to produce sub-logs that are compliant (or non-compliant) with some given LTL_f properties. We conducted empirical experiments and measured the execution times for performing the checking with RUM and DECLARE4PY. The findings of these experiments demonstrate that both solutions can execute complex tasks in reasonable time.

2. Tool description

We present here a Java solution and a Python solution for checking LTL_f properties in event logs.

The Java solution is provided in the form of a GUI-based functionality implemented in RUM. RUM is purposely designed to provide an easy, intuitive and user-friendly interface, making it suitable to be used by both process mining experts and non-experts [5]. The tool's graphical interface plays an important role in supporting the use of $SLTL_f$ and DECLARE for conformance checking and log filtering, by providing options to build chains of filters, modify them, instantiate constraints from templates, import and export filter configurations, analyze the checking results and the corresponding statistics, export compliant and non-compliant traces in different sub-logs. The verification of the LTL_f formulas in RUM is based on the tree-based search presented in [3]. The graphical user interface represents the real novelty of the Java solution because it provides a user experience that is not provided by any existing tool for LTL_f checking. Another distinguishing feature of this solution resides in the possibility to check B-DECLARE constraints, which is not provided in other solutions. Finally, it is also important to note that all LTL_f properties that can be checked in RUM are treated as filters that can be combined with standard log filters (e.g., on trace durations, on attribute values, or on timestamps) to extract traces with specific characteristics from an event log.

The Python solution is provided in the DECLARE4PY API. It is intended to be used by researchers willing to integrate this functionality in new tools, to perform large experimentations as well to perform Machine Learning related tasks. DECLARE4PY represents the first Python API able to perform conformance checking based on $SLTL_f$ and (B-)DECLARE constraints. This

solution uses finite state automata for LTL_f checking built from the input formulas through a C++ engine¹. Furthermore, it also performs parallel executions that partition the event log into sub-logs that are assigned to different threads running in parallel.

3. Tool Maturity

The execution times of the LTL_f checkers were measured through a series of experiments performed using event logs extracted from four datasets widely used in the literature. These logs, available in the well-known XES standard,² have the characteristics shown in Table 1 and are identified with labels D1, D2, D3, and D4.

Table 1

Statistics of the datasets used in the experiments.

Event log	# traces	# events
repairexample.xes (D1)	500	5725
sepsiscases.xes (D2)	1000	4307
teleclaim.xes (D3)	2500	32962
road-traffic-fine-management-process.xes (D4)	5000	36095

The experiments were conducted by applying the $SLTL_f$ and the B-DECLARE patterns to the event logs. In the case of B-DECLARE, the template parameters were replaced by the disjunction of two and five activities (randomly chosen in each log). The experiments were performed five times and the average time values were computed. All experiments were performed on a MacBook Pro machine equipped with an Apple M1 processor with 8 cores. The obtained results are illustrated in Table 2.

The results show that, in RUM, the execution times grow linearly with the number of traces and the number of constraints in the reference model. This type of behavior was expected since larger logs and a higher number of constraints inherently require more processing time. In DECLARE4PY, we can observe mostly the same behavior apart from some cases in which the execution times remain constant. This is due to the fact that, in RUM, both the results of checking all constraints together and of checking each individual constraint are provided. Instead, DECLARE4PY only shows the results obtained by checking all constraints together. Consequently, in the Java solution, the system will always perform the check of all constraints on all traces. Instead, in DECLARE4PY, when a trace does not satisfy a constraint, that trace is not checked anymore. Therefore, if at some point all traces in the log do not satisfy the constraints checked so far, checking the other constraints do not increase the execution time since there are no traces to be evaluated.

The execution times for DECLARE4PY are generally higher wrt. RUM. This is due to the fact that the Python solution first builds automata from the input constraints and afterwards performs the checking. Since the automata construction is exponential in the formula [7], this adds an overhead in DECLARE4PY.

¹<https://github.com/whitemech/lydia>

²<https://www.xes-standard.org/openxes/start>

Table 2

Execution times (in seconds).

# constraints	RuM				DECLARE4Py			
	D1	D2	D3	D4	D1	D2	D3	D4
SLTL_f								
5	0.39	1.04	1.79	4.77	0.62	2.55	0.59	5.89
10	0.61	1.43	3.26	7.98	0.59	2.63	0.60	6.82
15	0.83	2.42	5.12	11.07	0.63	2.25	0.59	6.82
20	1.04	2.79	6.02	12.89	0.59	2.13	0.57	6.54
2B-DECLARE								
5	0.78	0.93	1.81	3.26	2.25	4.68	9.10	5.20
10	0.82	1.87	3.32	6.39	2.29	7.43	9.12	5.31
15	1.04	2.76	4.58	9.69	2.18	7.68	9.13	5.25
20	1.23	3.22	5.96	12.25	2.32	7.11	9.34	5.17
5B-DECLARE								
5	0.44	1.07	2.11	4.03	1.44	5.40	7.18	16.92
10	0.86	2.31	3.64	7.56	1.46	5.32	7.30	17.64
15	1.26	2.99	5.39	10.98	1.54	5.35	7.22	17.63
20	1.57	4.10	6.97	15.13	1.44	5.35	7.33	17.64

4. Screencasts and websites

The source code of RuM is available at <https://bitbucket.org/doorless1634/thesis/src/tedi-thesis/> and the source code of DECLARE4PY can be found at <https://github.com/ivanDonadello/Declare4Py/>. In the latter repository, several Jupiter notebook-based tutorials are available explaining different LTL_f checking use cases. The video presentation of this paper can be accessed at <https://www.youtube.com/watch?v=4wy3C1EfYJw>.

References

- [1] W. M. P. van der Aalst, *Process Mining - Data Science in Action*, Springer, 2016.
- [2] C. Di Ciccio, A. Marrella, A. Russo, Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches, *J. Data Semant.* 4 (2015) 29–57.
- [3] W. M. P. van der Aalst, H. T. de Beer, B. F. van Dongen, Process mining and verification of properties: An approach based on temporal logic, in: *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 130–147.
- [4] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, DECLARE: Full support for loosely-structured processes, in: *EDOC*, IEEE Computer Society, 2007, pp. 287–300.
- [5] A. Alman, C. Di Ciccio, D. Haas, F. M. Maggi, A. Nolte, Rule mining with RuM, in: *2nd International Conference on Process Mining, ICPM 2020*, 2020, pp. 121–128.
- [6] I. Donadello, F. Riva, F. M. Maggi, A. Shikhizada, Declare4Py: A python library for declarative process mining, in: *BPM (PhD/Demos)*, volume 3216 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 117–121.
- [7] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: *IJCAI, IJCAI/AAAI*, 2013, pp. 854–860.