Studies on the Discovery of Declarative Control Flows from Error-prone Data

Claudio Di Ciccio[1 2 ‡] and Massimo Mecella[1 †]

[1] Sapienza – Università di Roma, Rome, Italy
[†] mecella@dis.uniroma1.it
[2] Wirtschaftsuniversität Wien, Vienna, Austria
[‡] claudio.di.ciccio@wu.ac.at

Abstract. The declarative modeling of workflows has been introduced to cope with flexibility in processes. Its rationale is based on the idea of stating some basic rules (named constraints), tying the execution of some activities to the enabling, requiring or disabling of other activities. What is not explicitly prohibited by such constraints is implicitly considered legal, w.r.t. the specification of the process. Declarative models for workflows are based on a taxonomy of constraint templates. Constraints are thus instances of constraint templates, applied to specific activities. Many algorithms for the automated discovery of declarative workflows associate to each constraint a support. The support is a statistical measure assessing to what extent a constraint was respected during the enactment(s) of the process. In current state-of-the-art literature, constraints having a support below a user-defined threshold are considered not valid for the process. Thresholds are useful for filtering out guesses based on possible misleading events, reported in logs either because of errors in the execution, unlikely process deviations, or wrong recordings in logs. The latter circumstance can be considered extremely relevant when logs are not written down directly by machines reporting their work, but extracted from other source of information. Here, we present an insight on the actual capacity of filtering constraints by modifying the threshold for support, on the basis of real data. Then, taking a cue from the results performed on such analysis, we consider the trend of support when controlled errors are injected into the log, w.r.t. individual constraint templates. Through these tests, we demonstrate by experiment that each constraint template reveal to be less or more robust to different kinds of error, according to its nature.

Keywords: process mining, artful process, declarative workflow, noisy event log

1 Introduction

Processes are typically represented as graphs, delineating their possible executions altogether, from the beginning up to the end. Most of the used notations are indeed derived by Petri Nets [2], such as Workflow Nets [1], BPMN [8], YAWL

[4]. The classical approach is called “imperative” because it explicitly represents every step allowed by the process model at hand. This leads to the likely increase of graphical objects as the process allows more alternative executions. The size of the model, though, has undesirable effects on the understandability and also on the likelihood of errors (see [18] for an insight of the Seven Process Modeling Guidelines): larger models tend to be more difficult to understand [19], not to mention the higher error probability which they suffer from, with respect to small models [17].

The declarative workflow models [22] have been introduced to cope with flexibility in processes. Its rationale is based on the idea of stating some basic rules (named constraints), tying the execution of some activities to either the enabling, requiring or disabling of other activities. What is not explicitly prohibited by such constraints is implicitly considered legal, w.r.t. the specification of the process. Declarative models for workflows are based on a taxonomy of constraint templates. Constraints are thus instances of constraint templates, applied to specific activities. A collection of constraints constitute altogether a declarative workflow. ConDec [20], now renamed Declare, is the most used language for modeling declarative workflows in the community of Business Process Management. It provides an extendible list of constraint templates, which we will consider in the remainder of this paper. Declarative models are particularly effective with some non-conventional kinds of process. For instance, professors, researchers, information engineers and all those professionals contributing to the production of a valuable but intangible products, such as knowledge, are commonly defined “knowledge workers” [23]. They are used to dealing with rapid decisions among multiple choices, based on their expertise, competence and intuition. There is an art in the management of their work. This is the reason for the name assigned to their processes: artful processes [14], which belong to the larger category of knowledge-intensive processes [9]. Artful processes are thus very flexible, dynamic and subject to change. Due to their characteristics, the declarative approach suits to their modeling [5]. Mining their workflow would be of extreme interest for understanding the best practices and winning strategies adopted by expert knowledge workers.

Process Mining [3], a.k.a. *Workflow Mining* [2], is the set of techniques that allow the extraction of process descriptions, stemming from a set of recorded real executions. Such executions are intended to be stored in so called *event log*s, i.e., textual representations of a temporarily ordered linear sequence of tasks. Many techniques have been proposed for mining Declare workflows ([16,15,10,11,7,6]). Most of them associate to each discovered constraint a support, i.e., a statistical measure assessing to what extent a constraint was respected during the enactment(s) of the process. Those discovered constraints having a support below a user-defined threshold are considered not valid for the process. Thresholds are useful for filtering out guesses based on possible misleading events, reported in event logs either because of errors in the execution, or due to very unlikely process deviations, or caused by wrong registrations of events in logs. The latter circumstance can be considered extremely relevant when event logs are not written down directly by machines reporting their work, but extracted from other sources of information. Artful processes, e.g., are known to be scarcely automated [9]. Therefore, there are few possibilities to rely on classical system logs, keeping track of their executions. As a matter of fact, despite the advent of structured case management tools, many enterprise processes are still "run" over email messages. Artful processes, for instance, often require the collaboration of many actors, who usually share their information by means of email messages. Thus, email messages are a valuable source of information and event logs can be extracted out of them, relying on their content and meta-data (e.g., the delivery timestamp). [12] presents a novel approach and a tool, named MailOfMine, designed to mine declarative workflows for artful processes out of email collections. First, MailOfMine inspects subjects, bodies and headers of given archives of email messages: assuming that reading about the execution of an activity can be interpreted as the reporting of its actual enactment, it searches the email messages where one among a list of user-defined expressions is found. Each is considered an event. Then, considering the temporal ordering of email messages in every archive, a trace in the log is built accordingly. Such log is passed to the MailOfMine control flow discovery algorithm (MINERful), which returns the declarative model for the artful process laying behind the email communications analyzed. Extracting logs out of email messages leads to possible errors though, due to the automated interpretation of semi-structured texts. Hence, such extracted logs are intrinsically prone to errors. Thereby, mistakes in the discovered workflow are likely to increase.

This is actually the question we search an answer for in this paper: what happens to unknown models when they are discovered on the basis of logs which are affected by errors. [13] investigates an approach for repairing *process models* basing on event data. Conversely, we consider the possible unreliability of data which process models are discovered from, supposing that process models were not previously known at all. In this paper, we first report the analysis of the results obtained by applying MailOfMine to real data, focused on the precision of the inferred model with respect to the support threshold. Then, we present an insight on the trend of the support in presence of errors, injected into synthetic logs. We focus on different types of errors (insertion or deletion of events) and spreading policies (a given percentage per each trace or all over the log). We repeat our experiments for each of the possible constraint templates that the MINERful algorithm is able to discover. Thus, we aim at understanding the different levels of robustness that constraint templates show w.r.t. the different types of errors.

The remainder of the paper is as follows. Section 2 describes the constraint templates of Declare and their usage for describing a declarative process model. Section 3 reports the results of tests on real data (Section 3.1) and experiments conducted on the basis of tunable injection of errors into synthetic logs (Section 3.2). Section 4 concludes this paper and outlines the future paths for our investigation that this paper sheds light on.

2 The declarative process model

Here we abstract activities as symbols (e.g., ρ, σ) of an alphabet Σ, appearing in finite strings, which, in turn, represent process traces. We will interchangeably use the terms "activity", "character" and "symbol", as well as "trace" and "string", then. We adopt the subset of Declare taxonomy of constraints for modeling processes, as in [16]. For a comprehensive analysis of all the constraint templates in Declare, the reader can refer to [20,21].

Constraints are temporal rules constraining the execution of activities. E.g., *Response*(ρ, σ) is a constraint on the activities ρ and σ, forcing σ to be executed if the ρ activity was completed before. Such rules are meant to adhere to specific constraint *templates*. *RespondedExistence* is the template of *RespondedExistence*(ρ, σ). We further categorize constraint templates into *constraint types*. For instance, *RespondedExistence* belongs to the *RelationConstraint* type. Figure 1 depicts the subsumption hierarchy of Declare constraints.

Declare constraints are always referred to an activity at least, which we call "implying": if it is executed, the constraint is triggered – vice-versa, if it does not appear in the trace, the constraint has no effect on the trace itself. The *Existence*(M, ρ) constraint imposes ρ to appear at least M times in the trace. We rename *Existence*$(1, \rho)$ as *Participation*(ρ). The *Absence*(N, ρ) constraint holds if ρ occurs at most $N - 1$ times in the trace. We call *Absence*$(2, \rho)$ as *Uniqueness*(ρ). *Init*(ρ) makes each trace start with ρ.

The aforementioned constraints fall under the type of *ExistenceConstraint*s, as they relate to an "implying" activity only. The following are named *RelationConstraint*s, since the execution of the implying imposes some conditions on another activity, namely the "implied".

RespondedExistence(ρ, σ) holds if, whenever ρ is read, σ was either already read or going to occur (i.e., no matter if before or afterwards). Instead, *Response*(ρ, σ) enforces it by requiring a σ to appear after ρ, if ρ was read. *Precedence*(ρ, σ) forces σ to occur after ρ as well, but the condition to be verified is that σ was read - namely, you can not have any σ if you did not read a ρ before. *AlternateResponse*(ρ, σ) and *AlternatePrecedence*(ρ, σ) strengthen respectively *Response*(ρ, σ) and *Precedence*(ρ, σ) by stating that *each* ρ (σ) must be followed (preceded) by at least one occurrence of σ (ρ). The "alternation" is in that you can not have two ρ's (σ's) in a row before σ (after ρ). *ChainResponse*(ρ, σ) and *ChainPrecedence*(ρ, σ), in turn, specialize *AlternateResponse*(ρ, σ) and *AlternatePrecedence*(ρ, σ), both declaring that no other symbol can occur between ρ and σ. The difference between the two is in that the former is verified for each occurrence of ρ, the latter for each occurrence of σ. The reader should note that the hierarchy under the *Precedence* constraint template does not inherit the base and implied symbols from the *RespondedExistence* parent; it overrides them both by inverting the two, instead. This is due to the semantics of the constraints themselves.

The *MutualRelation* constraints follow: they are verified iff two *RespondedExistence* (or descendant) constraints (resp., (*forward* and *backward*, in Figure 1) are satisfied. *CoExistence*(ρ, σ) holds if both *RespondedExistence*(ρ, σ) and *RespondedExistence*(σ, ρ) hold. *Succession*(ρ, σ) is valid if *Response*(ρ, σ) and *Precedence*(ρ, σ) are verified. The same holds with *AlternateSuccession*(ρ, σ), equivalent to the conjunction of *AlternateResponse*(ρ, σ) and *AlternatePrecedence*(ρ, σ), and *ChainSuccession*(ρ, σ), with respect to *ChainResponse*(ρ, σ) and *ChainPrecedence*(ρ, σ).

Finally, we consider *NegativeRelation* constraints: they are satisfied iff the related *MutualRelations* (*negated*, in Figure 1) are not. *NotChainSuccession*(ρ, σ) expresses the impossibility for σ to occur immediately after ρ (the opposite of *ChainSuccession*(ρ, σ)). *NotSuccession*(ρ, σ) generalizes the previous by imposing that, if ρ is read, no other σ can be read until the end of the trace (*Succession*(ρ, σ) is the *negated* constraint). *NotCoExistence*(ρ, σ) is even more restrictive: if ρ appears, not any σ can be in the same trace (the contrary of *CoExistence*(ρ, σ)).

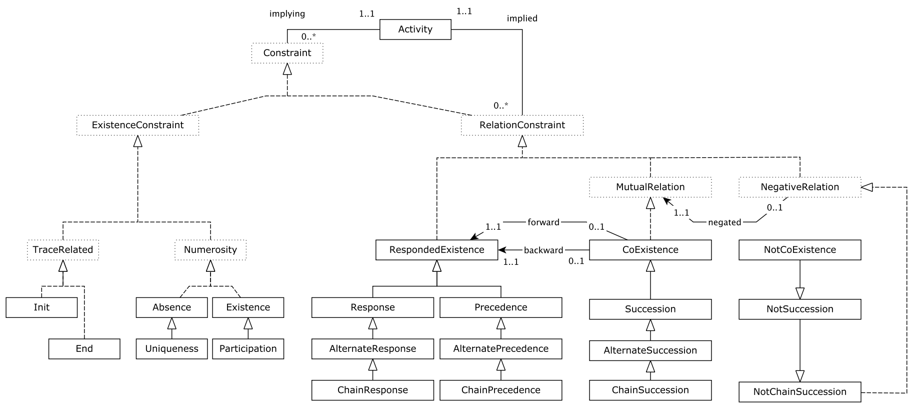


Fig. 1: The declarative process model's hierarchy of constraints. Taking into account the UML Class Diagram graphical notations, the Generalization ("is-a") relationship represents the subsumption between constraint templates. The subsumed is on the tail, the subsuming on the head. The Realization relationships indicate that the constraint template (and the subsumed in the hierarchy) belong to a specific type. Constraint templates are drawn as solid boxes, whereas the constraint types' boxes are dashed.

As a brief example, we may want to model the process of defining an agenda for a research project meeting. The schedule is discussed by email among the participants. We suppose that a final agenda will be committed ("confirm" – n) after that requests for a new proposal ("request" – r), proposals themselves ("propose" – p) and comments ("comment" – c) have been circulated.

The aforementioned activities are bound to the following constraints, then. If a request is sent, then a proposal is expected to be prepared afterwards (cf. *Response*(r, p)). Comments can be given in order to review a proposed agenda, or for soliciting the formulation of a new proposal. Thus, the presence of c in the trace is constrained to the presence of p (cf. *RespondedExistence*(c, p)). A confirmation is supposed to be mandatorily given after the proposal, and vice-versa any proposal is expected to precede a confirmation (cf. *Succession*(p, n)). We suppose the confirmation to be the *final* activity (cf. *End*(n)). This mandatory task (cf. *Participation*(n)) is not expected to be executed more than once (cf. *Uniqueness*(n)).

Hence, the example process consists in the six aforementioned constraints: *Response*(r, p), *RespondedExistence*(c, p), *Succession*(p, n), *Participation*(n), *Uniqueness*(n) and *End*(n). As an example, the following traces would be compliant to the workflow: pn, pcn, rpcn, rpcpn, rrpcrpcrcpcn, rpprpcccrpcn.

3 Experiments and evaluation

In order to inspect the quality of the control flow discovery in presence of error-prone logs, we first verified the whole MailOfMine system on real data (Section 3.1). There, data were extracted from the mailbox of an authors' colleague, known to be an expert in the area of the process to discover. As usual for artful processes, the process behind the analyzed email messages was not known a priori. Therefore, we could not apply an automated comparison between the resulting workflow model and the originating process, since no definition for the originating process was available at all. Thus, the expert was requested to analyze and assess the discovered workflow model by categorizing the mined constraints. Being real data, the presence of errors in the phase of the extraction of event logs out of email messages was not tunable.

Thereafter, we created synthetic logs, where errors of different kinds were injected into event logs. Every event log was created as adhering to the specification of declarative processes comprising a single constraint at a time. For each log, i.e., a different constraint template was considered. Being known a priori the only constraint to be considered valid, when mined out of the synthetic log, we focused on the trend of its support, in order to monitor the robustness of the template w.r.t. given types of errors. We outline the results of that analysis in Section 3.2.

3.1 A real case study

As real data to conduct the experiments on, we took 6 mailbox IMAP folders containing email messages which concerned the management of 5 different European research projects (Figure 1a). Such folders belonged to a domain expert. Our aim was to use MailOfMine in order to discover the artful process of managing European research projects and validate the result, together with him.

In order to ease the revision process of the gathered results, we restricted the number of activities for the process to discover to 13. 8.998% of the total amount of email messages were considered related to the execution of an activity. The setup and the results of the inspection of email messages for extracting a log is quantitatively summarized in Table 1b. The log was passed to the control flow discovery algorithm, which returned a process model comprising c.a. 200 constraints. Each was verified to hold true within the log and associated to a support exceeding the user-defined threshold of 80%.

	Mailbox						
	1	2	3	4	5	6	Total
Messages	3523	39	844	4 746	1 479	60	8 770

(a) The input

Activities	13	Noticeably right discovered constraints	14 (6.422%)
Traces	6	Right discovered constraints	173 (69.725%)
Events	139	Wrong discovered constraints	45 (20.642%)
Discovered constraints	218	Utterly wrong discovered constraints	7 (3.211%)

(b) Retrieved information and mined constraints

Table 1: Evaluation of MailOfMine on real data: setup and results

In order to assess the validity of the mined process, we checked every constraint with the expert. This allowed us for a quantitative evaluation.For each constraint in the list, we asked him whether it was either: *(i)* right, i.e., it made sense with respect to his experience; *(ii)* noticeably right, i.e., it not only made sense but also suggested some surprising mechanisms in the workflow; *(iii)* wrong, i.e., not necessarily corresponding to reality; *(iv)* utterly wrong, i.e., not corresponding to reality, unreasonable. The last level was assigned to quite few constraints (7 out of 218), a half of how many were considered noticeably right (14). The model is not known a priori, but the expert could classify as right or wrong a guessed constraint. Then, the analysis helped us find only true positives (TP, i.e., right or noticeably right) and false positives (FP, i.e., wrong or utterly wrong). As a matter of fact, such situation of partial knowledge of the workflow reproduces a real case, where the artful process had not ever been formalized before.
Recalling that $Precision = \frac{TP}{TP+FP}$, the algorithm was proven to obtain a *Precision* degree of 0.794 over the real case study. Table 1b summarizes the encouraging results of this real case study evaluation. More than 75% of the constraints inferred were compliant to a realistic model of the process. Figure 2 shows the trend of true positives, false positives and overall (i.e., the sum of the preceding) constraints found, scaled in percentage by their total amount, with respect to their support. The quantities on the ordinates are cumulative, i.e., they represent the sum of the values which are gained up to the current abscissa. The curves show how, as the support increases, the distance between the cumulated false positives and the true positives grow. A line puts in evidence where the relative percentage of confirmed constraints overtakes the wrong, i.e., a "breakpoint" after which the rate of hits, in terms of accepted guesses, is higher than the rate of misses, in terms of wrong guesses. Such breakpoint corresponds to a support value of 0.85 (i.e., 5% higher than the threshold established a priori), which is little enough to limit the number of true positives below that soil to less than 10%. The same graph, although, depicts that more than 90% of errors are given a support exceeding that soil as well. Thus, shifting the threshold altogether would not lead to significant improvements in the quality of the returned process. Hence, we studied the trend of support for error-injected logs, taking into account and isolating the behavior of every constraint template to different types of errors.

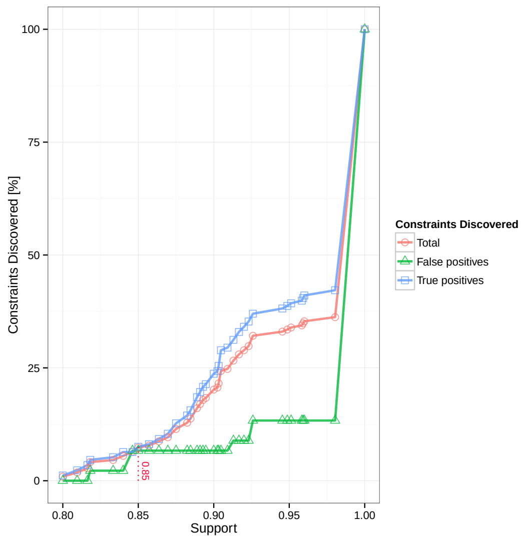


Fig. 2: The trend of the cumulative sum of constraints discovered, scaled by their total amount, w.r.t. the assigned support

3.2 Experiments over artificial error-injected logs

In order to test the robustness of MINERful with respect to the presence of errors in logs, we built an additional testing module, which injected a controlled amount of noise in the sequences of traces.

We identified three possible *types of error injection*:

1. insertion of spurious events in the log;
2. deletion of events from the log;
3. random insertion/deletion of events.

The errors were spread according to a given percentage[3]. The tester could also specify whether errors had to refer to a given activity, or not. In the latter case, every insertion or deletion was applied to an event picked each time at random.

In order to define how many errors had to be injected, and where, a *spreading policy* was requested too. It could be either:

1. to calculate the number of errors to inject w.r.t. the whole log, and distribute the error injections accordingly, or
2. to calculate the number of errors to inject w.r.t. every single trace, case by case.

In the latter case, every trace was made affected by a number of errors, computed on the basis of the number of target events in that trace. This reproduces a systematic error, taking place in every registered enactment of the process. In the former, some traces could remain untouched.

[3] In case the calculated number of errors to inject resulted in a non-integer number, the actual amount of errors was rounded up to the next integer (e.g., 0.2 was rounded to 1 error to inject).

Thereupon, we conducted an extensive analysis on the reaction of MINERful, the control flow discovery algorithm of MailOfMine, through an experiment set up as summarized in Table 2.

Activities (target)	8 (1)	Spreading policies	3
Generating constraints	18	Error types	3
Trace length	$[0, 30]$	Runs per combination	50
Log size	1 000	Error injection percentage	$[0, 30]$
		Total runs	167 400

Table 2: Setup of the experiments for monitoring the reaction of MINERful to the controlled error injection into logs

We created 18 groups of 9 300 synthetic logs each. Every group was generated so to comply to one constraint at a time, among the 18 templates involving a, as the implying activity, and (optionally) b, as the implied (i.e., *Participation*(a), *Uniqueness*(a), ..., *RespondedExistence*(a, b), *Response*(a, b), ...). The alphabet comprised 6 more non-constrained activities (c, d, ..., h), totalling 8. We chose a as the target activity for the injection of errors. Then, we injected errors in the synthetic logs, with all of the possible combinations of the aforementioned parameters (*(i)* insertion, deletion or random error type, *(ii)* over-string or over-collection spreading policy, *(iii)* error injection percentage ranging between 0 and 30%) and ran the control flow discovery algorithm of MailOfMine on the resulting altered logs. We collected the results and, for each of the 18 groups of logs, analyzed the trend of the support for the generating constraint. I.e., we looked at how the support for the only constraint which had to be verified all over the log lowered, w.r.t. the increasing percentage of errors injected. We also hightlighted those other constraints whose topmost computed support exceeded the value of 0.75[4], being them the most likely candidates to be false positives in the discovery.

The analysis of within-trace error-injected logs revealed to be more effective in stressing the resilience of constraints with respect to certain types of errors. In other words, it showed the structural weaknesses of constraint templates w.r.t. some types of error even for small percentages of injected errors. For instance, the support of *End*(a)'s (Figure 3) is not affected by the insertion of spurious a's in the traces (see Figure 3a), whereas it suffers from deletions of a's (Figure 3b).

In Section 2 we described the mechanism tying *MutualRelation* constraints to *forward* and *backward*-related constraints, as in the case of *AlternateSuccession* w.r.t. *AlternateResponse* and *AlternatePrecedence*. Then, here we remark that since *(i)* the support for *AlternateResponse*(a, b) remains unchanged in case of spurious inserted a's (Figure 4a), but not in case of deleted a's (Figure 4b), whilst

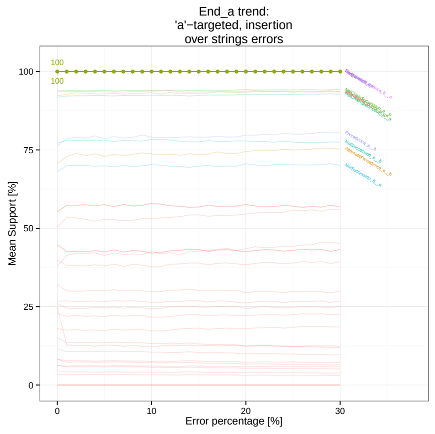
[4] We recall that assigning a constraint the support of 0.5 would be equivalent to asserting that such constraint would hold if, tossing a coin, a cross was shown in the end. Thereby, 0.75 is the least value of the topmost half of the "reliable" range.

(ii) conversely, the support for *AlternatePrecedence*(a, b) remains unchanged in case of deleted a's (Figure 4c), but not in case of inserted spurious a's (Figure 4d), *AlternateSuccession* inherits the sensitivity towards errors of both, resulting in a decreasing support for both faulty insertions and deletions of a's (Figure 5).

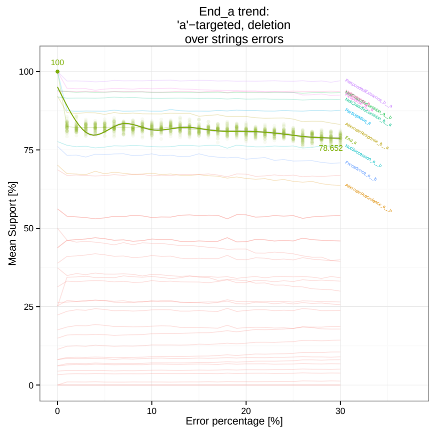
The analysis of over-collection error-injected logs showed smoother changes in curves, since errors are spread on a wider area of appearances, for the targeted activity. Therefore, it reveals a more realistic trend for the assessment of discovered constraints in presence of errors. We reasonably expect to have sparse errors in logs, rather than a fixed percentage of faults for every trace, as a matter of fact.

Along a branch in the constraints hierarchy (see Figure 1), we expect that the more a constraint is restrictive, the more its support decreases in terms of deviations from the expected behavior. We can prove it by evidence in, e.g., Figure 6, where the curve's slope gets steeper as we analyze the subsumed constraints along the *MutualRelation* constraints (i.e., *CoExistence*, *Succession*, *AlternateSuccession*, *ChainSuccession*).

The interested reader can download the whole collection of graphs depicting the gathered results at the following address:
`http://www.dis.uniroma1.it/~cdc/code/minerful/latest/errorinjectiontestresults.zip`

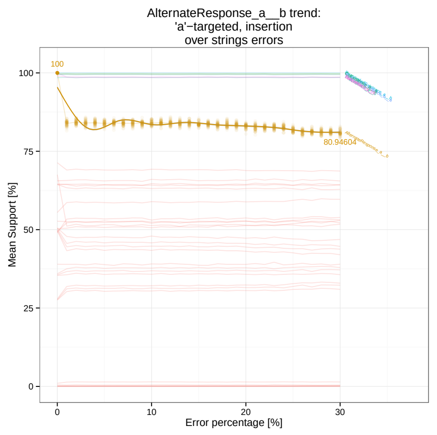


(a) The trend of the support for *End*(a), w.r.t. the percentage of spurious events inserted into every string

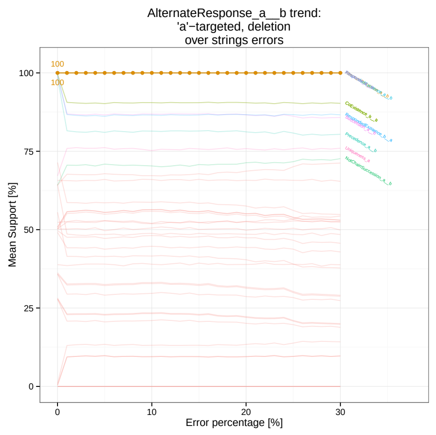


(b) The trend of the support for *End*(a), w.r.t. the percentage of events deleted from every string

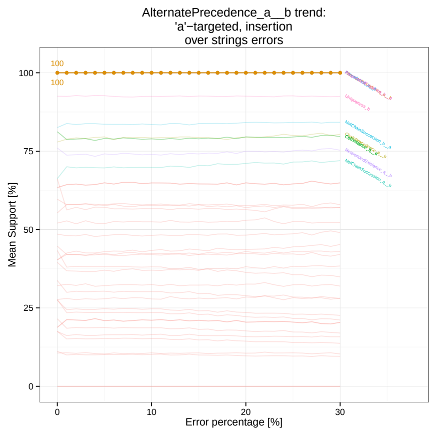
Fig. 3: The trend of the support for *End*, w.r.t. the errors injected in the log, within every trace



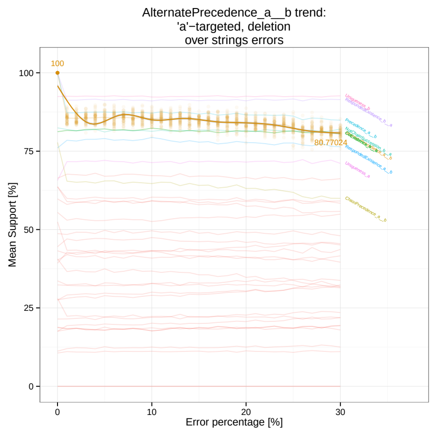
(a) The trend of the support for $\mathit{AlternateResponse}(\mathsf{a},\mathsf{b})$, w.r.t. the percentage of spurious events inserted into every string



(b) The trend of the support for $\mathit{AlternateResponse}(\mathsf{a},\mathsf{b})$, w.r.t. the percentage of spurious events inserted into every string

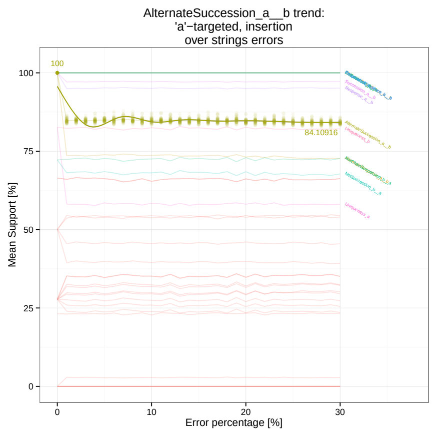


(c) The trend of the support for $\mathit{AlternatePrecedence}(\mathsf{a},\mathsf{b})$, w.r.t. the percentage of events deleted from every string

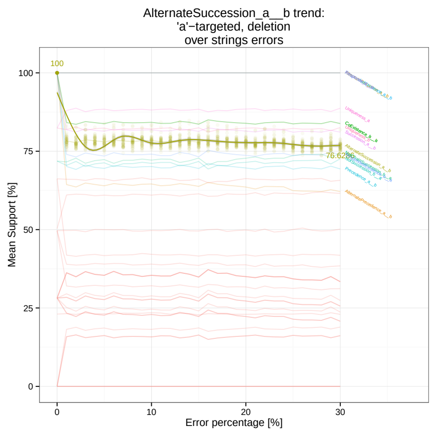


(d) The trend of the support for $\mathit{AlternatePrecedence}(\mathsf{a},\mathsf{b})$, w.r.t. the percentage of events deleted from every string

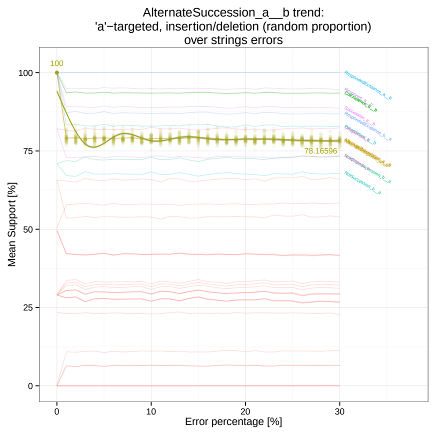
Fig. 4: The trend of the support for *AlternateResponse* and *AlternatePrecedence*, w.r.t. the errors injected in the log. The error injection policies under exam are both the insertion and deletion of a events, within each trace.



(a) The trend of the support for *AlternateSuccession*(a, b), w.r.t. the percentage of spurious events inserted into every string

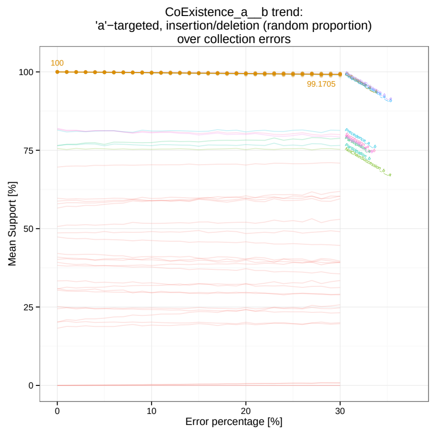


(b) The trend of the support for *AlternateSuccession*(a, b), w.r.t. the percentage of events deleted from every string

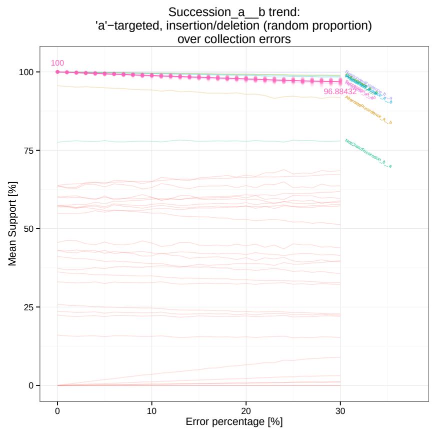


(c) The trend of the support for *AlternateSuccession*(a, b), w.r.t. the percentage of both deletions and insertions, applied to every string

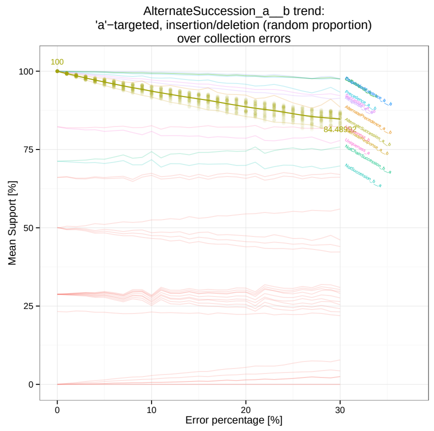
Fig. 5: The trend of the support for *AlternateSuccession*, w.r.t. the errors injected in the log, within each trace.



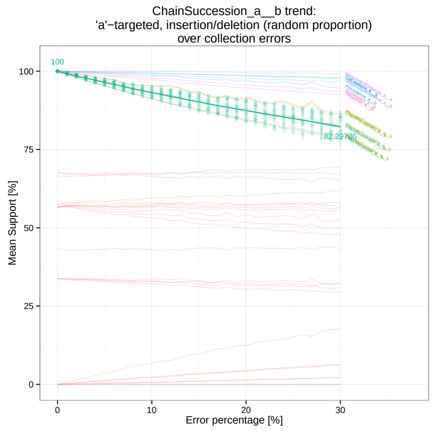
(a) The trend of the support for *CoExistence*(a, b), w.r.t. the percentage of both event deletions and insertions, spread over the whole log



(b) The trend of the support for *Succession*(a, b), w.r.t. the percentage of both event deletions and insertions, spread over the whole log



(c) The trend of the support for *AlternateSuccession*(a, b), w.r.t. the percentage of both event deletions and insertions, spread over into the whole log



(d) The trend of the support for *ChainSuccession*(a, b), w.r.t. the percentage of both event deletions and insertions, spread over into the whole log

Fig. 6: The trend of the support for the *MutualRelation* constraints, w.r.t. the errors injected in the log. The error injection policy under exam is the random insertion/deletion of a events, over the whole log.

4 Conclusions

Throughout this paper, we have analyzed the problem of discovering declarative workflows out of event logs which are affected by errors. To this aim, we first assessed the quality of a model, mined out of real data. We used a single threshold level for the estimated support of discovered constraints, in order to determine whether they could be considered valid for the mined process or not. The obtained results suggested that adjusting the level of such threshold did not considerably enhance the quality of the mined process altogether. Therefore, for each constraint in the set of Declare templates, we investigated the trend of its own estimated support with respect to the amount of errors injected into logs. By means of experiments carried out on synthetic data, we showed that the semantics of constraint templates actually affect their degree of robustness w.r.t. the presence or spurious events or the absence of expected ones in the log.

Starting from these results, we will investigate the problem of defining an automated approach for the self-adjustment of user-defined thresholds, on the basis of the nature of each discovered constraint. Intuitively, indeed, a more "robust" constraint should be considered valid in the log (and therefore for the process) if its support exceeds a higher threshold. On the contrary, the threshold should be diminished for more "sensitive" ones. We also aim at mixing such an approach with the analysis of different metrics, pertaining to the number of times an event occurred in the log. The intuition is that the more an event is frequent in the log, the less it can be considered subject to errors. Such metrics have been already considered in literature ([15]) for assessing the relevance of discovered constraints. We want to exploit them for estimating the reliability of constraints in mined processes as well.

References

1. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN. Lecture Notes in Computer Science, vol. 1248, pp. 407–426. Springer (1997).
2. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998).
3. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011).
4. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. 30(4), 245–275 (2005).
5. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009).
6. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. ACM Trans. Comput. Log. 9(4), 29:1–29:43 (August 2008).
7. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. T. Petri Nets and Other Models of Concurrency 2, 278–295 (2009).

8. Decker, G., Dijkman, R.M., Dumas, M., García-Bañuelos, L.: The business process modeling notation. In: ter Hofstede, A.M., van der Aalst, W.M.P., Adamns, M., Russell, N. (eds.) Modern Business Process Automation, pp. 347–368. Springer (2010).
9. Di Ciccio, C., Marrella, A., Russo, A.: Knowledge-intensive processes: An overview of contemporary approaches. In: ter Hofstede, A.H., Mecella, M., Sardina, S., Marrella, A. (eds.) KiBP. vol. 861, pp. 33–47. CEUR Workshop Proceedings (06 2012).
10. Di Ciccio, C., Mecella, M.: Mining constraints for artful processes. In: Abramowicz, W., Kriksciuniene, D., Sakalauskas, V. (eds.) BIS. Lecture Notes in Business Information Processing, vol. 117, pp. 11–23. Springer (05 2012).
11. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: CIDM. IEEE (04 2013).
12. Di Ciccio, C., Mecella, M., Scannapieco, M., Zardetto, D., Catarci, T.: MailOfMine – analyzing mail messages for mining artful collaborative processes. In: Aberer, K., Damiani, E., Dillon, T. (eds.) Data-Driven Process Discovery and Analysis, Lecture Notes in Business Information Processing, vol. 116, pp. 55–81. Springer (10 2012).
13. Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Barros, A.P., Gal, A., Kindler, E. (eds.) BPM. Lecture Notes in Computer Science, vol. 7481, pp. 229–245. Springer (2012).
14. Hill, C., Yates, R., Jones, C., Kogan, S.L.: Beyond predictable workflows: Enhancing productivity in artful business processes. IBM Systems Journal 45(4), 663–682 (2006).
15. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative process models from event logs. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE. Lecture Notes in Computer Science, vol. 7328, pp. 270–285. Springer (2012).
16. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: CIDM. pp. 192–199. IEEE (2011).
17. Mendling, J., Neumann, G., van der Aalst, W.M.P.: Understanding the occurrence of errors in process models based on metrics. In: Meersman, R., Tari, Z. (eds.) CoopIS. Lecture Notes in Computer Science, vol. 4803, pp. 113–130. Springer (2007).
18. Mendling, J., Reijers, H.A., van der Aalst, W.M.P.: Seven process modeling guidelines (7PMG). Information & Software Technology 52(2), 127–136 (2010).
19. Mendling, J., Reijers, H.A., Cardoso, J.: What makes process models understandable? In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM. Lecture Notes in Computer Science, vol. 4714, pp. 48–63. Springer (2007).
20. Pesic, M.: Constraint-based Workflow Management Systems: Shifting Control to Users. Ph.D. thesis, Technische Universiteit Eindhoven (10 2008), `http://repository.tue.nl/638413`
21. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: EDOC. pp. 287–300. IEEE Computer Society (2007).
22. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: Meersman, R., Tari, Z. (eds.) CoopIS. Lecture Notes in Computer Science, vol. 4803, pp. 77–94. Springer (2007).
23. Warren, P., Kings, N., Thurlow, I., Davies, J., Buerger, T., Simperl, E., Ruiz, C., Gomez-Perez, J.M., Ermolayev, V., Ghani, R., Tilly, M., Bösser, T., Imtiaz, A.: Improving knowledge worker productivity - the Active integrated approach. BT Technologiy Journal 26(2), 165–176 (2009).