A Two-Step Fast Algorithm for the Automated Discovery of Declarative Workflows

Claudio Di Ciccio
SAPIENZA Università di Roma, Italy
Email: cdc@dis.uniroma1.it

Massimo Mecella
SAPIENZA Università di Roma, Italy
Email: mecella@dis.uniroma1.it

***Abstract*—Declarative approaches are particularly suitable for modeling highly flexible processes. They especially apply to artful processes, i.e., rapid informal processes that are typically carried out by those people whose work is mental rather than physical (managers, professors, researchers, engineers, etc.), the so called "knowledge workers". This paper describes MINERful^{++}, a two-step algorithm for an efficient discovery of constraints that constitute declarative workflow models. As a first step, a knowledge base is built, with information about temporal statistics gathered from execution traces. Then, the statistical support of constraints is computed, by querying that knowledge base. MINERful^{++} is fast, modular, independent of the specific formalism adopted for representing constraints, based on a probabilistic approach and capable of eliminating the redundancy of subsumed constraints.**

I. INTRODUCTION

For a long time, structured business processes, such as the ones of public administrations, hospitals, insurance/financial institutions, etc., have been the main subject of workflow related research. The mainstream management tools, and process mining techniques, model processes with a graphical syntax derived from a subset of Petri Nets, that are Workflow Nets (WfN), explicitly designed to represent the control-flow dimension of a process. They draw the entire life-cycle of a process in terms of paths over graph-based structures, from the beginning to the end.

Managers, researchers, and all of the so called "knowledge workers" [1] are used to perform difficult tasks, which require complex, rapid decisions among multiple possible strategies, in order to fulfill specific goals. Such tasks constitute a kind of very flexible workflows, which change over time and according to contingencies: the so called "artful" business processes ([2], [3]). The need for flexibility in the definition of artful business processes leads to an alternative to the classical "imperative" approach: the "declarative" [4]. Rather than using a procedural language for expressing the allowed sequences of activities, the description of workflows is based on the usage of constraints. A *process scheme* (or *process* for short) turns into a semi-structured set of activities, where the semi-structuring connective tissue is represented by the set of constraints stating the interleaving rules among activities. Constraints do not force the tasks to follow a tight sequence, but rather leave them the flexibility to follow different paths, though respecting a set of rules that avoid illegal or non-consistent states in the execution. DecSerFlow and ConDec, now unified under the name of Declare [5], define constraints templates for declarative workflows as formulations of Linear Temporal Logic.

Process Mining [6], also referred to as *Workflow Mining*, is the set of techniques that allow the extraction of process descriptions, stemming from a set of recorded real executions (*traces*). ProM [7] is one of the most used plug-in based software environments for implementing workflow discovery techniques.

Many different techniques have been proposed for mining imperative processes, each addressing specific issues: *(i)* pure algorithmic (e.g., α algorithm [8] and its evolution α^{++} [9]); *(ii)* heuristic (e.g., [10]), able to handle frequencies of events, in order to filter out sporadic patterns; *(iii)* genetic (e.g., [11]), adopting an evolutionary approach to the discovery (hence, its computation evolves in a non-deterministic way); *(iv)* two-step (e.g., [12]). Heuristic and genetic algorithms have been introduced to cope with noise, that the pure algorithmic techniques were not able to manage. A smart extension to the previous research was achieved by the two-steps algorithm proposed in [12].It splits the computation in two phases: *(i)* the tunable mining of a Transition System (TS) representing the process behavior and *(ii)* the automated construction of a Petri Net bisimilar to the TS [13]. The tuning in the first phase let the expert decide whether the mining had to be either more strictly adhering or more permissive with respect to the behavior of the analyzed log, thus balancing between "overfitting" and "underfitting".

[14] presents an algorithm for mining Declare processes, implemented in ProM. The technique is based on the translation of Declare constraints into automata,which the log is replayed on top of. [15] describes the usage of inductive logic programming techniques to mine models expressed as a set of $\mathcal{S}$CIFF [16] Integrity Constraints, finally translated to the ConDec [17] notation. [18] extends this technique by weighting in a second phase the discovered constraints with a probabilistic estimation. [19] also proposes an evolution of [14] throughout a two-phase approach. The first phase makes use of the Apriori algorithm, in order to identify correlated activity sets. The candidate constraints are computed on the basis of the correlated activity sets only. During the second phase, the candidate constraints are checked as in [14]. In the end, only those mined costraints that comply to specific user-defined metrics are shown.

Here, we present the MINERful^{++} technique, which is a workflow discovery algorithm, able to infer declarative models out of traces gathered from the execution of enacted processes. Its peculiarities are: *(i)* modularity, i.e., it is based on two

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

steps, where the first builds a knowledge base for the second, effectively verifying the constraints as the results of specific queries over that; *(ii)* independence from the specific formalism adopted for representing constraints; *(iii)* probabilistic approach to the inference of constraints; *(iv)* capability of eliminating the redundancy of subsumed constraints.

The remainder of this paper is as follows: Section II summarizes declarative workflow constraints, adopted in relevant state-of-the-art work as well as in ours; Section III describes the MINERful^{++} technique in detail. Experiments on the proposed algorithm are presented in Section IV, along with some considerations about MINERful^{++} wrt. the techniques of [14] and [19]. Finally, Section V concludes the paper.

The interested reader can download the implementation of MINERful^{++}, together with the source code and the data for the experiments here presented, at the following address: http://www.dis.uniroma1.it/~cdc/code/minerful/latest/MINERful.zip.

II. Specification of constraints

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 Declare taxonomy of constraints for modeling processes, as in [14]. In the following, we briefly summarize the constraint templates that Declare is based upon. The reader can find further information in [5], [14]. Figure 1 depicts the subsumption hierarchy of Declare constraints. This will be exploited in the algorithm explained further in this paper (Section III-D).

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 $Participation(\rho)$. The *Absence*(N, ρ) constraint holds if ρ occurs at most $N - 1$ times in the trace. We call $Absence(2, \rho)$ as $Uniqueness(\rho)$. $Init(\rho)$ makes each trace start with ρ.

The aforementioned constraints fall under the category 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" (see Figure 1).

$RespondedExistence(\rho, \sigma)$ holds if, whenever ρ is read, σ was either already read or going to occur (i.e., no matter if before or afterwards). Instead, $Response(\rho, \sigma)$ enforces it by requiring a σ to appear after ρ, if ρ was read. $Precedence(\rho, \sigma)$ 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. $AlternateResponse(\rho, \sigma)$ and $AlternatePrecedence(\rho, \sigma)$ strengthen respectively $Response(\rho, \sigma)$ and $Precedence(\rho, \sigma)$ 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 ρs (σs) in a row before σ (after ρ). $ChainResponse(\rho, \sigma)$ and $ChainPrecedence(\rho, \sigma)$, in turn, specialize $AlternateResponse(\rho, \sigma)$ and $AlternatePrecedence(\rho, \sigma)$, 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.

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

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

In Table I, the semantics of Declare constraints are reported for sake of clarification. Semantics are expressed by means of regular expressions. This translation has been useful to the automated generation of synthetic traces, complying to a given model, which the proposed algorithm could be tested on top of (see Section IV). For sake of brevity, there we used the POSIX standard shortcuts. Therefore, in addition to the known Kleene star ($*$), alternation ($|$) and concatenation ( ) operators, we make use here of *(i)* the `.` and `[^x]` shortcuts for respectively matching any character in the alphabet, or any character but x, *(ii)* the `+` and `?` operators for respectively matching from one to any, or none to one, occurrences of the preceding expression, and *(iii)* the `{n,m}` notation, where `n` (resp. `m`) denotes the minumum (maximum) number of repetitions of the preceding pattern. Examples are provided so to give a hint on the sense of such constraints. The underlined characters are the "implying" symbols. Strongly emphasized characters are those checked in order to verify the constraint on the string.

A. An Example

Here we outline a brief example (cf. [20]). We 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 tasks and activities are bound to the following constraints (cf. Process Description 1).

If a request is sent, then a proposal is expected to be prepared afterwards (cf. $Response(\mathsf{r}, \mathsf{p})$). The presence of comments, in case, is due to a delay in the presentation of an expected proposal, or as a review of the previous.

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

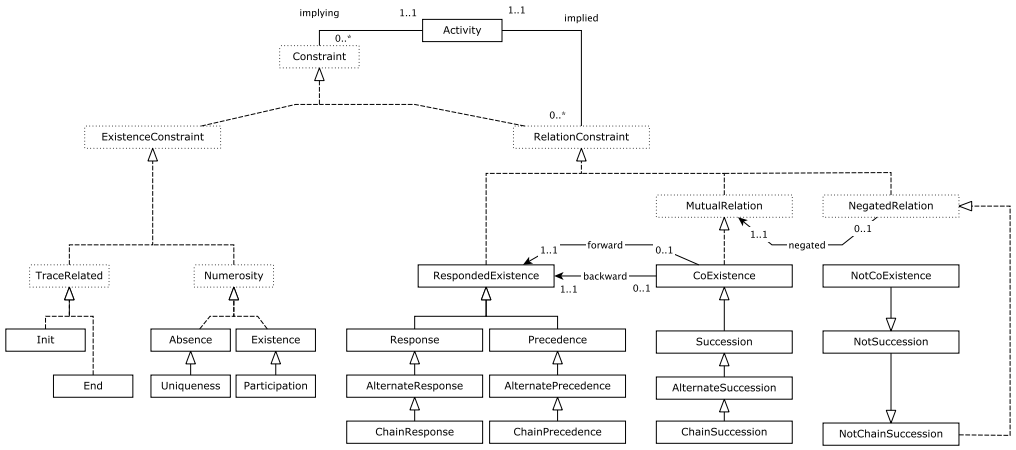


Fig. 1: The declarative process model constraints hierarchy

Constraint	Regular expression	Example
Existence constraints		
$Existence(n, a)$	`[^a]*(a[^a]*){n,}+[^a]*`	
$Participation(a) \equiv Existence(1, a)$	`[^a]*(a[^a]*)+[^a]*`	bcaac
$Absence(m+1, a)$	`[^a]*(a[^a]*){0,m}+[^a]*`	
$Uniqueness(a) \equiv Absence(2, a)$	`[^a]*(a)?[^a]*`	bcac
$Init(a)$	`a.*`	accbbbaba
$End(a)$	`.*a`	bcaaccbbbaba
Relation constraints		
$RespondedExistence(a, b)$	`[^a]*((a.*b)|(b.*a))*[^a]*`	bcaaccbbbaba
$Response(a, b)$	`[^a]*(a.*b)*[^a]*`	bcaaccbbbab
$AlternateResponse(a, b)$	`[^a]*(a[^a]*b)*[^a]*`	bcaccbbbab
$ChainResponse(a, b)$	`[^a]*(ab[^a^b]*)*[^a]*`	bcabbbab
$Precedence(a, b)$	`[^b]*(a.*b)*[^b]*`	caaccbbbaba
$AlternatePrecedence(a, b)$	`[^b]*(a[^b]*b)*[^b]*`	caaccbaba
$ChainPrecedence(a, b)$	`[^b]*(ab[^a^b]*)*[^b]*`	cababa
$CoExistence(a, b)$	`[^a^b]*((a.*b)|(b.*a))*[^a^b]*`	bcaccbbbaba
$Succession(a, b)$	`[^a^b]*(a.*b)*[^a^b]*`	caaccbbbab
$AlternateSuccession(a, b)$	`[^a^b]*(a[^a^b]*b)*[^a^b]*`	caccbab
$ChainSuccession(a, b)$	`[^a^b]*(ab[^a^b]*)*[^a^b]*`	cabab
Negative relation constraints		
$NotChainSuccession(a, b)$	`[^a]*(a[^a^b][^a]*)*([^a]*|a)`	bcaaccbbbba
$NotSuccession(a, b)$	`[^a]*(a[^b]*)*[^a^b]*`	bcaacca
$NotCoExistence(a, b)$	`[^a^b]*((a[^b]*)|(b[^a]*))?`	caacca

TABLE I: Semantics of Declare constraints as regular expressions

Thus, the presence of c in the trace is constrained to the presence of p (cf. $RespondedExistence(\mathsf{c}, \mathsf{p})$). A confirmation is supposed to be mandatorily given after the proposal, and vice-versa any proposal is expected to precede a confirmation (cf. $Succession(\mathsf{p}, \mathsf{n})$). We suppose the confirmation to be the *final* activity (cf. $End(\mathsf{n})$). This mandatory task (cf. $Participation(\mathsf{n})$) is not expected to be executed more than once (cf. $Uniqueness(\mathsf{n})$).

Process Description 1 The example process

$Response(\mathsf{r}, \mathsf{p})$

$RespondedExistence(\mathsf{c}, \mathsf{p})$

$Succession(\mathsf{p}, \mathsf{n})$

$Participation(\mathsf{n})$, $Uniqueness(\mathsf{n})$, $End(\mathsf{n})$

As an example, the following traces would be compliant to the given model: pn, pcn, rpcn, rpcpn, rrpcrpcrcpcn, rpprpcccrpcn.

III. The MINERful^{++} algorithm

MINERful^{++} is our proposed algorithm for mining declarative constraints out of finite traces of symbols (logs). MINERful^{++} is based on the concept of *MINERfulKB*: it holds all of the useful information extracted from the given traces and tailored to the further discovery of constraints that might lay behind. The first step of MINERful^{++} is to build this knowledge base (Section III-C), in order to easily infer the declarative model on top of it (Section III-D), during the second step. The final output is thus a set of constraints, proved to be valid on the the knowledge base. Each constraint is weighted with its *support*, i.e., a value ranging from 0 to 1, calculated as the normalized fraction of cases in which the constraint is verified over the set of input traces (see Table III).

A. Definitions

The MINERfulKB $\mathcal{K}$ is an interpretation for the MINERful interplay and the MINERful ownplay. This interpretation is

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

given by the application of MINERful^{++} to a collection of strings T, as described in the following Section III-B.

For the definition of MINERful interplay, we have to keep in mind that it is referred to a couple of symbols: one is considered as the *pivot*, $\rho \in \Sigma$, the other is the *searched*, $\sigma \in \Sigma$. For the definition of MINERful ownplay, only the *pivot* ρ matters, since it is focused on the statistics referred to a single activity only. For sake of simplicity, examples will consider a as the pivot ρ and b as the searched σ, over an alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. For sake of readability, we put input characters as indexes in the subscript of the function symbols. We also make use of the *distance* metric, which here represents the number of characters between ρ and σ. It assumes a positive value if σ *follows* ρ, negative if σ *precedes* ρ.

Definition 1 (MINERful interplay): A tuple $\mathcal{D} = \langle \delta, b^{\rightarrow}, b^{\leftarrow} \rangle$ where:

- $\delta_{\rho,\sigma}(d)$ $\delta : \Sigma \times \Sigma \times \mathbb{Z} \rightarrow \mathbb{N}^1$ is the *distances* function, mapping a distance $d \in \mathbb{N}^1$ between the pivot $\rho \in \Sigma$ and the searched $\sigma \in \Sigma$ to the number of cases they appeared at distance d in a trace (e.g., $\delta_{\mathsf{a},\mathsf{b}}(2) = 4$ means that we have the evidence of a searched b appearing 2 characters after the pivot a, as in the strings cacbcc, acbcacba and acbaaa); we recall that $\mathbb{N}^1$ is the set of natural integers excluding zero;
- $b^{\rightarrow}_{\rho,\sigma}$ $b^{\rightarrow} : \Sigma \times \Sigma \rightarrow \mathbb{N}$ is the *in-between onwards appearances* function, counting the number of cases where the pivot ρ was read again between ρ itself and the next searched σ (e.g., if $b^{\rightarrow}_{\mathsf{a},\mathsf{b}} = 2$, it means that the pivot a appeared 2 times between the preceding occurrence of a and the following first occurrence of the searched b, as in the substring accaacb);
- $b^{\leftarrow}_{\rho,\sigma}$ $b^{\leftarrow} : \Sigma \times \Sigma \rightarrow \mathbb{N}$ is the *in-between backwards appearances* function, counting the number of cases where the pivot ρ was read again between ρ itself and the searched σ, reading the string contrariwise (e.g., if $b^{\leftarrow}_{\mathsf{a},\mathsf{b}} = 3$, it means that the pivot a appeared 3 times between the following occurrence of a and the preceding last occurrence of the searched b, as in the substring bcacaaca);

With a slight abuse of notation, we consider in the following $\delta_{\rho,\sigma}(+\infty)$ and $\delta_{\rho,\sigma}(-\infty)$ to denote the number of cases in which the searched σ, respectively, did not appear in a string *after* the pivot ρ, or did not appear in a string *before* ρ. $\delta_{\rho,\sigma}(0)$ represents the number of cases in which the searched σ did not appear, either before or after ρ, in the string.

Definition 2 (MINERful ownplay): A tuple $\mathcal{E} = \langle \gamma, g^i, g^l \rangle$ where

- $\gamma_\rho(n)$ $\gamma : \Sigma \times \mathbb{N} \rightarrow \mathbb{N}$ is the *global appearances* function, mapping the pivot ρ and a natural number $n \in \mathbb{N}$ to the number of traces in which ρ was read n times (e.g., $g_{\mathsf{a}}(4) = 2$ means that it happened to the pivot a to be read exactly four times in two strings only in the log, as in aabbabccaaabab and babacaa in the analyzed collection of traces);
- g^i_ρ $g^i : \Sigma \rightarrow \mathbb{N}$ represents the number of strings where the pivot ρ appeared as the *initial* symbol (e.g., if $g^i_{\mathsf{a}} = 5$, five traces started with a);
- g^l_ρ $g^l : \Sigma \rightarrow \mathbb{N}$ represents the number of strings where the pivot ρ appeared as the *last* symbol (e.g., if $g^l_{\mathsf{a}} = 0$, no trace ended with a);

As an example, let us suppose to interpret the MINERfulKB over a singleton $T = \{\mathsf{aabbac}\}$. Then, for what a is concerned, (the portion of) $\mathcal{D}$ is shown in Table II, and (the portion of) $\mathcal{E}$ is as it follows:

$$\left\langle \gamma_{\mathsf{a}}(n) = \left\{ \begin{array}{ll} 1 & n = 3 \\ 0 & n \in \mathbb{N} \setminus \{3\} \end{array} \right\}, g^i_{\mathsf{a}} = 1, g^l_{\mathsf{a}} = 0 \right\rangle$$

B. The algorithm

Algorithm 1 The MINERful^{++} pseudo-code algorithm (bird-eye watching)

```
K ← COMPUTEKBONWARDS(T, Σ)
K ← COMPUTEKBBACKWARDS(T, Σ)
B ← DISCOVERCONSTRAINTS(K, Σ, |T|)
```

Algorithm 1 presents a bird-eye view of the technique. The different steps will be detailed in the following sections.

Algorithm 2 The pseudo-code of the DISCOVERCONSTRAINTS algorithm

```
Require: τ = 1.0    # An optional user-defined threshold
1: procedure DISCOVERCONSTRAINTS(K, Σ, |T|)
2:     B ← CALCSUPPORTFORCONSTRAINTS(K, Σ, |T|)
3:     B ← CLEANOUTPUT(B, Σ)
4:     B ← FILTEROUTPUTBYTHRESHOLD(B, τ) return B
5: end procedure
```

C. Construction of the MINERfulKB

The input is a collection of strings, called T, and an alphabet, Σ. At the end of the run of this first phase, we have the interpretations for both the MINERful interplay and the MINERful ownplay, i.e., $\mathcal{K}$, on the basis of T.

The MINERfulKB is designed in order to be tailored to the further reasoning for constraints discovery. Thus, the latter step becomes easier and faster, than analyzing it directly from the raw data (the collection of strings). At the same time, building the MINERfulKB has to be reasonably fast: moving the whole complexity to that operation would take no advantage to the overall technique.

The details about the algorithm for building the MINERfulKB are detailed in [21], together with the proof of the following Lemma.

Lemma 1: The procedure for building the knowledge base of MINERful^{++} is *(i)* linear time w.r.t. the number of traces in the log, *(ii)* quadratic time w.r.t. the size of traces in the log, *(iii)* quadratic time w.r.t. the size of the alphabet. Defined t_{max} as the longest trace in the log, the complexity is therefore $O(|T| \cdot |t_{max}|^2 \cdot |\Sigma|^2)$ [21].

As an intuition, we recall here that the nature of the MINERfulKB itself allows us to specify an algorithm which is completely on-line, i.e., it refines the MINERfulKB as new

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

	$-\infty$	$\cdots$	-4	-3	-2	-1	0	$+1$	$+2$	$+3$	$+4$	$+5$	$\cdots$	$+\infty$
$\delta_{\mathsf{a,b}}$	2	0	0	0	1	1	0	1	2	1	0	0	0	0
$\delta_{\mathsf{a,c}}$	3	0	0	0	0	0	0	1	0	0	1	1	0	0

$b^{\rightarrow}_{\mathsf{a,b}} = 1;$ $b^{\leftarrow}_{\mathsf{a,b}} = 0$

$b^{\rightarrow}_{\mathsf{a,c}} = 2;$ $b^{\leftarrow}_{\mathsf{a,c}} = 0$

TABLE II: Examples of MINERfulKB, interpreted over aabbac

strings occur *(i)* and new characters in the string are read, with no need to go back on already processed data in the end. For each symbol picked up from the string, the temporary data structures related to the occurrences of the already appeared characters are updated *(ii)*. Finally, every activity is linked to specific information to every other activity in the alphabet *(iii)*, within the knowledge base.

D. Discovery of Constraints

Declarative processes are modeled by a set of constraints, imposing the rules that each process instance must follow, whatever the execution trace is. The set of mined constraints is listed in Table III. The listed functions allow to assess the validity of a constraint on the basis of the respective support[1].

Here we call "support" the value, ranging from 0 to 1, that represents the normalized fraction of cases in which the constraint is verified, over the set of traces T. Such functions are all based on mathematical operations performed on data coming from the MINERfulKB only, plus the information about the size of T, i.e., how many strings were read.

Algorithm 3 The pseudo-code of the CALCSUPPORTFORCONSTRAINTS procedure

```
1:  procedure CALCSUPPORTFORCONSTRAINTS(𝒦, Σ, |T|)
2:      for all ρ ∈ Σ do
3:          if Γ_ρ > 0 then
4:              for all ExCon ⊑ ExistenceConstraint do
5:                  ℬ ← ℬ ∪ {ExCon, calc(ExCon, ρ)}
6:              end for
7:              for all σ ∈ Σ do
8:                  for all ReCon ⊑ RelationConstraint do
9:                      ℬ ← ℬ ∪ {ReCon, calc(ReCon, ρ)}
10:                 end for
11:             end for
12:         end if
13:     end for return ℬ
14: end procedure
```

For sake of brevity, although, here we also define:

$$\Gamma_\rho = \sum_{n>0} \gamma_\rho(n) \cdot n$$

i.e., the total number of appearances of ρ in the whole set of traces T.

Rather than indicating the exact number of times a task can be done (in a range from the lower to the upper), so to specify that all of the $Existence(n, \rho)$ constraints hold, for n ranging from 0 to $\min_{\gamma_\rho(n)>0} n$ (and dually consider $Absence(m+1, \rho)$ valid for each m from $\max_{\gamma_\rho(m)>0} m$ onwards), we preferred to introduce a looser couple of constraints, stating whether a task ρ must be executed ($Participation(\rho)$) or not, and whether it must not be done more than once ($Uniqueness(\rho)$). We believe that providing the minimum and the maximum for ranges would have been for artful processes too overfitting, when mined, or too restrictive, when enacted.

Table III shows the functions used in order to compute the support for the constraints, with respect to the MINERfulKB.

Algorithm 4 The pseudo-code of CLEANOUTPUT

```
1:  procedure CLEANOUTPUT(ℬ, Σ)
2:      𝒞 := clone ℬ
3:      for all ⟨c, s_c⟩ ∈ 𝒞 do
4:          if RelationConstraint(c) then
5:              if hasParent(𝒞, c) then
6:                  ⟨p, s_p⟩ := getParent(𝒞, c)
7:                  if s_p ⩽ s_c then
8:                      ℬ ← ℬ \ {⟨p, s_p⟩}
9:                  else
10:                     ℬ ← ℬ \ {⟨c, s_c⟩}
11:                 end if
12:             end if
13:         end if
14:         if MutualRelation(c) then
15:             ⟨f, s_f⟩ := getForward(𝒞, c)
16:             ⟨r, s_r⟩ := getBackward(𝒞, c)
17:             if s_f < s_c ∧ s_r < s_c then
18:                 ℬ ← ℬ \ {⟨f, s_f⟩}
19:                 ℬ ← ℬ \ {⟨r, s_r⟩}
20:             end if
21:         end if
22:         if NegatedRelationConstraint(c) then
23:             ⟨n, s_n⟩ := getNegated(𝒞, c)
24:             if s_p ⩽ s_c then
25:                 ℬ ← ℬ \ {⟨n, s_n⟩}
26:             else
27:                 ℬ ← ℬ \ {⟨c, s_c⟩}
28:             end if
29:         end if
30:     end for return ℬ
31: end procedure
```

The overall DISCOVERCONSTRAINTS algorithm is presented in Algorithm 2. It consists of three procedure calls. The first, CALCSUPPORTFORCONSTRAINTS, stores in a bag, namely $\mathcal{B}$, the output of each function listed before in Table III (Algorithm 3). $\mathcal{B}$ is a collection of tuples $\langle b, s_b \rangle$, each associating to a constraint b the related support, s_b. We want to focus the attention here on line 3. The condition put there

[1]Please note that [3], [21] presents a collection of predicates asserting whether a constraint is verified or not, on top of the MINERfulKB. Conversely, here we propose functions for evaluating the statistical support of constraints.

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

Constraint	Support function	Constraint	Support function
$Existence(n,a)$	$1-\frac{\sum_{i=0}^{n-1}\gamma_a(i)}{\lvert T\rvert}$	$Participation(a)$	$1-\frac{\gamma_a(0)}{\lvert T\rvert}$
$Absence(m,a)$	$\frac{\sum_{i=0}^{m}\gamma_a(i)}{\lvert T\rvert}$	$Uniqueness(a)$	$\frac{\gamma_a(0)+\gamma_a(1)}{\lvert T\rvert}$
$Init(a)$	$\frac{g_a^i}{\lvert T\rvert}$	$End(a)$	$\frac{g_a^l}{\lvert T\rvert}$
$RespondedExistence(a,b)$	$1-\frac{\delta_{a,b}(0)}{\Gamma_a}$		
$Response(a,b)$	$1-\frac{\delta_{a,b}(+\infty)}{\Gamma_a}$	$Precedence(a,b)$	$1-\frac{\delta_{b,a}(-\infty)}{\Gamma_b}$
$AlternateResponse(a,b)$	$1-\frac{b^{\rightarrow}_{a,b}+\delta_{a,b}(+\infty)}{\Gamma_a}$	$AlternatePrecedence(a,b)$	$1-\frac{\delta_{b,a}(-\infty)+b^{\leftarrow}_{a,b}}{\Gamma_b}$
$ChainResponse(a,b)$	$\frac{\delta_{a,b}(1)}{\Gamma_a}$	$ChainPrecedence(a,b)$	$\frac{\delta_{b,a}(-1)}{\Gamma_b}$
$CoExistence(a,b)$	$1-\frac{\delta_{a,b}(0)+\delta_{b,a}(0)}{\Gamma_a+\Gamma_b}$	$NotCoExistence(a,b)$	$\frac{\delta_{a,b}(0)+\delta_{b,a}(0)}{\Gamma_a+\Gamma_b}$
$Succession(a,b)$	$1-\frac{\delta_{a,b}(+\infty)+\delta_{b,a}(-\infty)}{\Gamma_a+\Gamma_b}$	$NotSuccession(a,b)$	$\frac{\delta_{a,b}(+\infty)+\delta_{b,a}(-\infty)}{\Gamma_a+\Gamma_b}$
$AlternateSuccession(a,b)$	$1-\frac{b^{\rightarrow}_{a,b}+\delta_{a,b}(+\infty)+b^{\leftarrow}_{b,a}+\delta_{b,a}(-\infty)}{\Gamma_a+\Gamma_b}$		
$ChainSuccession(a,b)$	$\frac{\delta_{a,b}(1)+\delta_{b,a}(-1)}{\Gamma_a+\Gamma_b}$	$NotChainSuccession(a,b)$	$1-\frac{\delta_{a,b}(1)+\delta_{b,a}(-1)}{\Gamma_a+\Gamma_b}$

TABLE III: Functions computing the support of constraints

avoids characters never appeared in the log to be the base for any inferred constraint. Given that *ex falso quod libet*, a character that was never read might be declared as supporting each constraint, though it would be senseless to the mining purpose, as it would add no bit of information to the gathered knowledge. In order to filter the irrelevant constraints out of the output, we make use of two methods, the aim of which is: *(i)* not to show trivially deducible constraints[2]; *(ii)* let the user decide a threshold of reliability, i.e., decide what is the least support for a constraint to be considered valid.

The former objective is managed by Algorithm 4, CLEANOUTPUT, which requires no user intervention. The latter is obtained by Algorithm 5, FILTEROUTPUTBYTHRESHOLD, which expects a parameter to be optionally provided by the user: $\tau \in [0,1]$, i.e., the threshold. By default, it is set to 1.0 – i.e., only those constraints that are verified over the whole log are shown. The block between lines 4 and 13 in Algorithm 4 involves every *RelationConstraint* (see the hierarchy in Figure 1). There, the constraints that are subsumed by others are removed, when they have less support. We explore one step in the hierarchy at a time because it is known by definition that the support monotonically decreases descending along the hierarchy.

The *MutualRelation* constraints are managed in the block from line 14 to line 21. In that case, if a *MutualRelation* constraint is known to have a support which is at least greater than *both* of the involved *RelationConstraint*s, the latter couple can be removed. Otherwise, no action is taken.

Finally, from line 22 to line 29, a selection between each *NegatedRelation* constraint and its *negated* is given, on the basis of the respective support.

The *hasParent*, *getParent*, *getForward*, *getBackward* and *getNegated* functions are reported but not explained in detail here. They explore the subsumptions and the associations between constraints as drawn in Figure 1: *hasParent* and *getParent* traverse the subsumption hierarchy, *getForward* and *getBackward* return the partner constraints, between *CoExistence* and the two related *RespondedExistence* (the same applies to the hierarchies below, as for *ChainSuccession* wrt. *ChainResponse* and *ChainPrecedence*), *getNegated* returns the *MutualRelation* constraint (like *CoExistence*) that is negated by the *NegatedRelation* constraint (like *NotCoExistence*). These functions do not depend on the interpretation of the MINERful interplay and the MINERful ownplay, but only on the semantics of constraints. E.g., it is known a priori that $getForward(\cdot, AlternateSuccession_{\rho,\sigma})$ returns $AlternateResponse_{\rho,\sigma}$ and its support. This is due to the semantics of the *Succession* constraint only, regardless the support it has or whatever the collection of traces T is.

[2]e.g., it is enough to say that $ChainPrecedence(a,b)$ holds, rather than explicitly return as valid constraints $ChainPrecedence(a,b)$, $AlternatePrecedence(a,b)$ and $Precedence(a,b)$ – where the latter couple is directly implied by the first: see Figure 1

Algorithm 5 The pseudo-code of the FILTEROUTPUTBYTHRESHOLD procedure

```
1: procedure FILTEROUTPUTBYTHRESHOLD(B, τ)
2:     for all ⟨b, s_b⟩ ∈ B do
3:         if ¬(|2 · τ − 1| ⩽ s_b ⩽ 1) then
4:             B ← B \ {⟨b, s_b⟩}
5:         else
6:             B ← B \ {⟨b, s_b⟩} ∪ {⟨b, (s_b − τ)/(1 − τ)⟩}
7:         end if
8:     end forreturn B
9: end procedure
```

The FILTEROUTPUTBYTHRESHOLD procedure (Algorithm 5) finally filters out those constraints whose support is out of a closed interval ranging around τ. The interval's maximum is always equal to 1.0, whereas its width is scaled according to the value of τ (see line 3). The support given in output is recalculated and shown with respect to the soil of τ, thus assuming either a negative or positive value, scaled to the distance from τ to 1.0 (line 6). From the description of the Algorithms 3, 4 and 5, the following Lemma 2 follows.

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

The proof is omitted for sake of space.

Lemma 2: The procedure for discovering the constraints of processes out of the MINERful^{++} knowledge base is *(i)* quadratic time w.r.t. the size of the alphabet, *(ii)* linear in the number of constraint templates, which is fixed and equal to 18 (thus constant); therefore, the complexity is $O(|\Sigma|^2)$.

From Lemmata 1 and 2, Theorem 1 follows.

Theorem 1: The MINERful^{++} is *(i)* linear time w.r.t. the number of strings in the log, *(ii)* quadratic time w.r.t. the size of strings in the log, *(iii)* quadratic time w.r.t. the size of the alphabet; therefore, the complexity is $O(|T| \cdot |t_{max}|^2 \cdot |\Sigma|^2)$.

IV. Experiments and evaluation

Experiments on the proposed technique have been conducted on both synthetic and real data. All of the tests were performed on a Sony VAIO VGN-FE11H (Intel Core Duo T2300 1.66 GHz, 2 MB L2 cache, with 2 GB of DDR2 RAM at 667 Mhz), having Ubuntu Linux 10.04 as the operating system and Java JRE v1.6. In order to produce synthetic logs, we first considered the example workflow, outlined in Section II-A. Random strings were created integrating our tool with Xeger[3], a Java open-source library for generating random text from regular expressions (see Table I), based on [22].

We tested the algorithm by varying the input in terms of alphabet size (different symbols appearing in the traces), number of constraints, range of the number of characters per string (see Setup 1 in Table V). The constraints ranged from a minimal set of four ($Unique(\mathsf{n})$, $Participation(\mathsf{n})$, $End(\mathsf{n})$, $Succession(\mathsf{p},\mathsf{n})$) to the maximal set of seven (including $Response(\mathsf{r},\mathsf{p})$, $RespondedExistence(\mathsf{c},\mathsf{p})$, $AlternatePrecedence(\mathsf{r},\mathsf{c})$)[4]. In order to consider the performances' degradation over increasing alphabets, we also executed a new set of experiments, according to Setup 2 (Table V).

Figure 2a shows the time taken by the algorithm to run, in comparison with the number of traces in the logs. The time taken for the algorithm to mine constraints, with respect to the average length of the strings is depicted in Figure 2b. There, the alphabet size is fixed and equal to 5. The dependency is quadratic as expected. The time taken for the algorithm to discover the workflow model, with respect to the size of the input (namely, the total number of events read) is depicted in Figure 2c. There, each curve correspond to a different number of activities in the log.

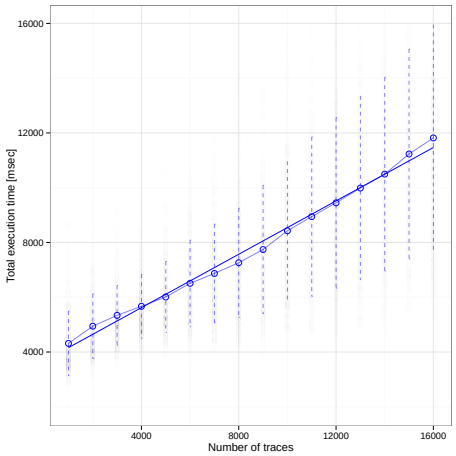
In order to test the efficiency of MINERful^{++} when dealing with real-life cases, we tested it with two well known benchmarks, taken from the latest Business Process Intelligence Challenges (BPIC) [23], [24]. Table IV reports some interesting results, taken from the experiments on both synthetic and real logs. For each test, we performed the analysis on Declare Miner [14], publicly available as a ProM plug-in[5]. We tuned Declare Miner so to check the same constraints MINERful^{++} discovers[6]. Table IV shows that the execution of MINERful^{++}

[3]http://code.google.com/p/xeger/

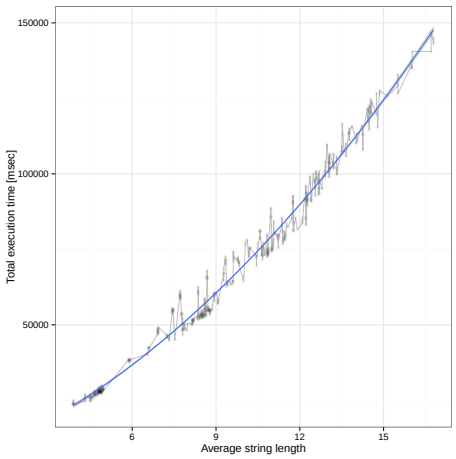
[4]The number of constraints involved in the generation of synthetic strings is proven not to affect performances, as shown in [3]

[5]http://www.win.tue.nl/declare/declare-miner/

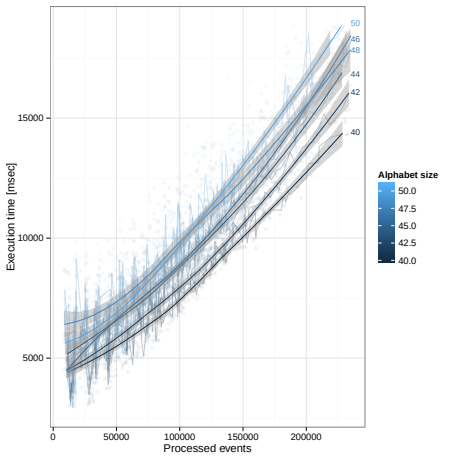
[6]PoE equal to 100%, PoIW equal to 0%, PoI equal to 100%, all constraints selected excluding those not covered in Table I.



(a) Time needed for the execution, w.r.t. the number of traces (from Setup 2 – see Table V): only the tests where the size of the alphabet is greater than 25 are considered



(b) Time needed for the execution, w.r.t. the string length (from Setup 1 – see Table V): only the tests where the size of the alphabet is equal to 5 are plotted



(c) Time needed for the execution, w.r.t. the number of symbols read (from Setup 2 – see Table V): each curve corresponds to a given size of the alphabet (here, only if comprising more than 40 symbols)

Fig. 2: Experimental results

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

Source	Tasks	Traces	Events processed		Total comp. time	Engine
Synth. log, Setup 1	5	100 000	1 676 447	(avg. 16.764)	00:00:15.036	MINERful^{++}
					00:11:20.000	Declare Miner [14]
Synth. log, Setup 2	52	16 000	296 277	(avg. 18.517)	00:00:25.111	MINERful^{++}
					00:21:24.000	Declare Miner [14]
Financial log [23]	24	13 087	262 200	(avg. 20.035)	00:00:08.997	MINERful^{++}
					00:08:54.000	Declare Miner [14]
Hospital log [24]	624	1 143	150 291	(avg. 131.488)	00:04:34.099	MINERful^{++}
					03:39:13.000	Declare Miner [14]

TABLE IV: Performances of MINERful^{++} over synthetic and real cases

Setup	Min. length	Max. length	Number of traces	Alphabet size	Total tests
1	[0,8]	[5,20]	$[10^2, 10^6]$	$[2, 5]$	29 000
2	[0,2]	[10,25]	$[10^3, 16 \cdot 10^3]$	$[5, 50]$	13 536

TABLE V: Setup of the experiments

is faster than Declare Miner. We recall here that Declare Miner could discover any LTL-expressible constraint, whereas MINERful^{++} is not customizable at this level from the user. Nonetheless, MINERful^{++} is completely unsupervised, as the user has not to select in advance the constraints that have to be checked. She is (optionally) requested to provide a threshold to filter out some loosely-supported constraints, just in the end, for sake of her ease to access the extracted information, with no impact on the performances of the algorithm.

V. Concluding remarks

As a concluding remark, we would like to highlight that the algorithm presented in this paper is part of a complex approach, aimed at discovering artful processes out of email messages. Through the application of other techniques, out of the scope of this paper, we are currently abstracting email threads as traces over an alphabet of symbols. Thereby, we are mining the declarative models of the processes laying behind the communications that those email messages convey [20] through MINERful^{++}.

Recently, [19] improved the performances of [14] thanks to its pre-processing phase. At this moment, we were not able to test it. The experimental results reported in [19] show that the performances could be likely comparable to the ones of MINERful^{++}, though. Metrics introduced in [19] (Confidence, Interest Factor and CPIR) can be adopted in MINERful^{++} too, being them all based on the shared notion of "support" of a constraint (see Section III-D).

Currently, we are extending the MINERfulKB so to make MINERful^{++} able to discover branched Declare constraints. We are also studying the robustness of MINERful^{++} through the controlled injection of errors in the input traces. Moreover, we are investigating a user-oriented approach to let users extend the set of discovered constraints, also beyond the templates of [14] and [5].

References

[1] P. Warren, N. Kings, I. Thurlow et al., "Improving knowledge worker productivity - the Active integrated approach," *BT Technologiy Journal*, vol. 26, no. 2, pp. 165–176, 2009.

[2] C. Hill, R. Yates, C. Jones, and S. L. Kogan, "Beyond predictable workflows: Enhancing productivity in artful business processes," *IBM Systems Journal*, vol. 45, no. 4, pp. 663–682, 2006.

[3] C. Di Ciccio and M. Mecella, "Mining constraints for artful processes," in Proc. *BIS 2012*.

[4] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D*, vol. 23, no. 2, pp. 99–113, 2009.

[5] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "Declare: Full support for loosely-structured processes," in Proc. *EDOC 2007*.

[6] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[7] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, A. Rozinat, E. Verbeek, and T. Weijters, "ProM: The process mining toolkit," in *BPM 2009 (Demos)*.

[8] W. M. P. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, 2004.

[9] L. Wen, W. M. P. van der Aalst, J. Wang, and J. Sun, "Mining process models with non-free-choice constructs," *Data Min. Knowl. Discov.*, vol. 15, no. 2, pp. 145–180, 2007.

[10] A. Weijters and W. van der Aalst, "Rediscovering workflow models from event-based data using little thumb," *Integrated Computer-Aided Engineering*, vol. 10, p. 2003, 2001.

[11] A. K. Medeiros, A. J. Weijters, and W. M. Aalst, "Genetic process mining: an experimental evaluation," *Data Min. Knowl. Discov.*, vol. 14, no. 2, pp. 245–304, 2007.

[12] W. van der Aalst, V. Rubin, H. Verbeek, B. van Dongen, E. Kindler, and C. Gnther, "Process mining: a two-step approach to balance between underfitting and overfitting," *Software and Systems Modeling*, vol. 9, pp. 87–111, 2010.

[13] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Deriving petri nets from finite transition systems," *Computers, IEEE Transactions on*, vol. 47, no. 8, pp. 859 –882, aug. 1998.

[14] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst, "User-guided discovery of declarative process models," in Proc. *CIDM 2011*.

[15] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, "Exploiting inductive logic programming techniques for declarative process mining," *T. Petri Nets and Other Models of Concurrency*, vol. 2, pp. 278–295, 2009.

[16] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni, "Verifiable agent interaction in abductive logic programming: The SCIFF framework," *ACM Trans. Comput. Log.*, vol. 9, no. 4, pp. 29:1–29:43, 2008.

[17] M. Pesic and W. M. P. van der Aalst, "A declarative approach for flexible business processes management," in Proc. *BPM Workshops 2006*.

[18] E. Bellodi, F. Riguzzi, and E. Lamma, "Probabilistic declarative process mining," in Proc. *KSEM 2010*.

[19] F. M. Maggi, R. P. J. C. Bose, and W. M. P. van der Aalst, "Efficient discovery of understandable declarative process models from event logs," in Proc. *CAiSE 2012*.

[20] C. Di Ciccio, M. Mecella, M. Scannapieco, D. Zardetto, and T. Catarci, "MailOfMine – analyzing mail messages for mining artful collaborative processes," in Post-proc. *SIMPDA 2011*.

[21] C. Di Ciccio and M. Mecella, "MINERful, a mining algorithm for declarative process constraints in MailOfMine," Tech. Rep., 2012. [Online]. Available: http://ojs.uniroma1.it/index.php/DIS_TechnicalReports/issue/view/416

[22] A. Møller. (2011, September) dk.bricks.automaton. Aarhus University. [Online]. Available: http://www.brics.dk/automaton/index.html

[23] B. F. van Dongen, "Real-life event logs – a loan application process," BPIC 2012. [Online]. Available: http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[24] ——, "Real-life event logs – a hospital log," BPIC 2011. [Online]. Available: http://dx.doi.org/10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/CIDM.2013.6597228

Errata Corrige

The list of regular expressions defining semantics of Declare constraint templates (published in Di Ciccio *et al.* [2012]; Di Ciccio and Mecella [2013]), did not cover the whole set of complying strings. Table A amends the preceding.

The authors want to thank to Michael Westergaard and Christian Stahl for their precious advice and fruitful collaboration.

Constraint	Regular expression	Example
Existence constraints		
Existence(*n*, a)	`[^a]*(a[^a]*){n,}+[^a]*`	
Participation(a) ≡ *Existence*(1, a)	`[^a]*(a[^a]*)+[^a]*`	bc**a**ac
Absence(*m* + 1, a)	`[^a]*(a[^a]*){0,m}+[^a]*`	
Uniqueness(a) ≡ *Absence*(2, a)	`[^a]*(a)?[^a]*`	bc**a**c
Init(a)	`a.*`	**a**ccbbbaba
End(a)	`.*a`	bcaaccbbbab**a**
Relation constraints		
RespondedExistence(a, b)	`[^a]*((a.*b.*)|(b.*a.*))*[^a]*`	**bca**accbbb**a**b**a**
Response(a, b)	`[^a]*(a.*b)*[^a]*`	bc**a**acc**b**bb**ab**
AlternateResponse(a, b)	`[^a]*(a[^a]*b[^a]*)*[^a]*`	bc**a**cc**b**bb**ab**
ChainResponse(a, b)	`[^a]*(ab[^a]*)*[^a]*`	bc**ab**bb**ab**
Precedence(a, b)	`[^b]*(a.*b)*[^b]*`	c**a**acc**b**b**a**ba
AlternatePrecedence(a, b)	`[^b]*(a[^b]*b[^b]*)*[^b]*`	c**a**acc**ba**b**a**
ChainPrecedence(a, b)	`[^b]*(ab[^b]*)*[^b]*`	c**ab**a**b**a
CoExistence(a, b)	`[^ab]*((a.*b.*)|(b.*a.*))*[^ab]*`	**bca**accbbb**a**b**a**
Succession(a, b)	`[^ab]*(a.*b)*[^ab]*`	c**a**acc**b**bb**ab**
AlternateSuccession(a, b)	`[^ab]*(a[^ab]*b[^ab]*)*[^ab]*`	c**a**cc**bab**
ChainSuccession(a, b)	`[^ab]*(ab[^ab]*)*[^ab]*`	c**abab**
Negative relation constraints		
NotChainSuccession(a, b)	`[^a]*(aa*[^ab][^a]*)*([^a]*|a)`	bc**a**acc**b**bbb**a**
NotSuccession(a, b)	`[^a]*(a[^b]*)*[^ab]*`	bc**a**acc**a**
NotCoExistence(a, b)	`[^ab]*((a[^b]*)|(b[^a]*))?`	c**a**acc**a**

Table A: Semantics of Declare constraints as regular expressions

References

Claudio Di Ciccio and Massimo Mecella. A two-step fast algorithm for the automated discovery of declarative workflows. In *4th IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2013, Singapore, April 16-19, 2013*, pages 135–142. IEEE, 2013.

Claudio Di Ciccio, Massimo Mecella, Monica Scannapieco, Diego Zardetto, and Tiziana Catarci. MailOfMine – analyzing mail messages for mining artful collaborative processes. In Karl Aberer, Ernesto Damiani, and Tharam Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 55–81. Springer, 2012.

This document is a pre-print copy of the manuscript
(Di Ciccio and Mecella 2013)
published by IEEE (available at ieeexplore.ieee.org).

The final version of the paper is identified by DOI: 10.1109/CIDM.2013.6597228

References

Di Ciccio, Claudio and Massimo Mecella (2013). "A Two-Step Fast Algorithm for the Automated Discovery of Declarative Workflows". In: *CIDM*. IEEE, pp. 135–142. ISBN: 978-1-4673-5895-8. DOI: 10.1109/CIDM.2013.6597228. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6588692.

BibTeX

```
@InProceedings{   DiCiccio.Mecella/CIDM2013:TwoStepFast,
  author        = {Di Ciccio, Claudio and Mecella, Massimo},
  booktitle     = {CIDM},
  title         = {A Two-Step Fast Algorithm for the Automated Discovery of
                   Declarative Workflows},
  year          = {2013},
  pages         = {135-142},
  publisher     = {IEEE},
  crossref      = {CIDM2013},
  doi           = {10.1109/CIDM.2013.6597228},
  keywords      = {process mining, declarative process}
}
@Proceedings{     CIDM2013,
  title         = {{IEEE} Symposium on Computational Intelligence and Data
                   Mining, {CIDM} 2013, Singapore, 16-19 April, 2013},
  year          = {2013},
  publisher     = {{IEEE}},
  isbn          = {978-1-4673-5895-8},
  url           = {http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6588692}
}
```