DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

MINERful, a Mining Algorithm for Declarative Process Constraints in MailOfMine

Claudio Di Ciccio
Massimo Mecella

Technical Report n. 3, 2012

MINERful, a Mining Algorithm for Declarative Process Constraints in MailOfMine

Claudio Di Ciccio, Massimo Mecella

SAPIENZA – *Università di Roma,*
Dipartimento di Ingegneria Informatica, Automatica e Gestionale ANTONIO RUBERTI
Via Ariosto 25, Roma, Italy

{cdc,mecella}@dis.uniroma1.it

***Abstract*— Artful processes are informal processes typically carried out by those people whose work is mental rather than physical (managers, professors, researchers, engineers, etc.), the so called "knowledge workers". MAILOFMINE is a tool, the aim of which is to automatically build, on top of a collection of e-mail messages, a set of workflow models that represent the artful processes laying behind the knowledge workers activities. After an outline of the approach and the tool, this paper focuses on the mining algorithm, able to efficiently compute the set of constraints describing the artful process. Finally, an experimental evaluation of it is reported.**

***Index Terms*— Process mining, artful process, declarative workflows.**

I. INTRODUCTION

For a long time, formal business processes (e.g., the ones of public administrations, of insurance/financial institutions, etc.) have been the main subject of workflow related research. Informal processes, a.k.a. "artful processes", are conversely carried out by those people whose work is mental rather than physical (managers, professors, researchers, engineers, etc.), the so called "knowledge workers" [1]. In contrast to business processes that are formal and standardized, often informal processes are not even written down, let alone defined formally, and can vary from person to person even when those involved are pursuing the same objective. Knowledge workers create informal processes "on the fly" to cope with many of the situations that arise in their daily work. Though informal processes are frequently repeated, they are not exactly reproducible even by their originators – since they are not written down – and can not be easily shared either. Their outcomes and their information exchanges are done very often by means of e-mail conversations, which are a fast, reliable, permanent way of keeping track of the activities that they fulfill.

Understanding artful processes involving knowledge workers is becoming crucial in many scenarios. Here we mention some of them:

- *personal information management (PIM)*, i.e., how to organize one's own activities, contacts, etc. through the use of software on laptops and smart devices (iPhones/iPads, smartphones, tablets). Here, inferring artful processes in which a person is involved allows the system to be proactive and thus drive the user through its own tasks (on the basis of the past) [2];
- *information warfare*, especially in supporting anti-crime intelligence agencies: let us suppose that a government bureau is able to access the e-mail account of a suspected person. People planning a crime or an act out of law are used to speak a language of their own to express duties and next moves, where meanings may not match with the common sense. Though, a system should build the processes that lay behind their communications anyway, exposing the activities and the role of the actors. At that point, translating the sense of misused words becomes an easier task for investigators, and allows inferring the criminal activities of the suspected person(s);
- *enterprise engineering*: in design and engineering, it is important to preserve more than just the actual documents making up the product data. Preserving the "soft knowledge" of the overall process (the so-called product life-cycle) is of critical importance for knowledge-heavy industries. Hence, the idea here is to take to the future not only the designs, but also the knowledge about processes, decision making, and people involved [3], [4], [5].

The objective of the MAILOFMINE approach, firstly introduced in [6], is to automatically build, on top of a collection of e-mail messages, a set of workflow models that represent the artful processes laying behind the knowledge workers activities. Here, we outline the general approach and enter into the detail of the process mining algorithm, able to infer constraints (indeed our approach is based on a declarative specification of process models). Then we present some experiments, showing the validity and efficiency of the technique.

The work here presented is related to the so called *process mining*, a.k.a. *workflow mining* [7], that is the set of techniques allowing the extraction of structured process descriptions, stemmed from a set of recorded real executions (stored in the *event log*s). ProM [8] is one of the most used plug-in based software environment for implementing workflow mining techniques. Most of the mainstream process mining tools model processes as Workflow Nets (WFNs – see [9]), explicitly designed to represent the control-flow dimension of a workflow. From [10] onwards, many techniques have been proposed, in order to address specific issues: pure algorithmic (e.g., α algorithm [11] and its evolution α^{++} [12]), heuristic (e.g., [13]), genetic (e.g., [14]). Indeed, heuristic and genetic algorithms have been introduced to cope with noise, that the pure algorithmic techniques were not able to manage. A very smart extension to the previous research work has been recently achieved by the two-steps algorithm proposed in [15].

The need for flexibility in the definition of processes leads to an alternative to the classical "imperative": the "declarative" approach. Rather than using a procedural language for expressing the allowed sequences of activities, it is based on the description of workflows through the usage of constraints: the idea is that every task can be performed, except what does not respect them. Such constraints, in DecSerFlow [16] and ConDec [17] (now named Declare), are formulations of Linear Temporal Logic and have a graphical representation as well. [18] outlines an algorithm for mining Declare [18], [19], [20] processes, implemented in ProM. The technique is based on [21], [22], [20], for the translation of Declare constraints into automata, and [23], for the optimization of such task. [24] described the usage of inductive logic programming techniques to mine models expressed as a $\mathcal{S}$CIFF [25] theory, finally translated to the ConDec notation.

The technique introduced in this paper differs from both [24] and [18] in that it does not directly verify the candidate constraints over the whole set of traces in input. It prepares an ad-hoc knowledge base of its own, so to further analyze the response to specific queries.

We believe that the declaration of collaborative workflows constraints can be expressed by means of regular expressions, rather than LTL formulae: regular expressions express finite languages (i.e., processes with finite traces, where the number of enacted tasks is limited). LTL formulae are thought to be used for verifying

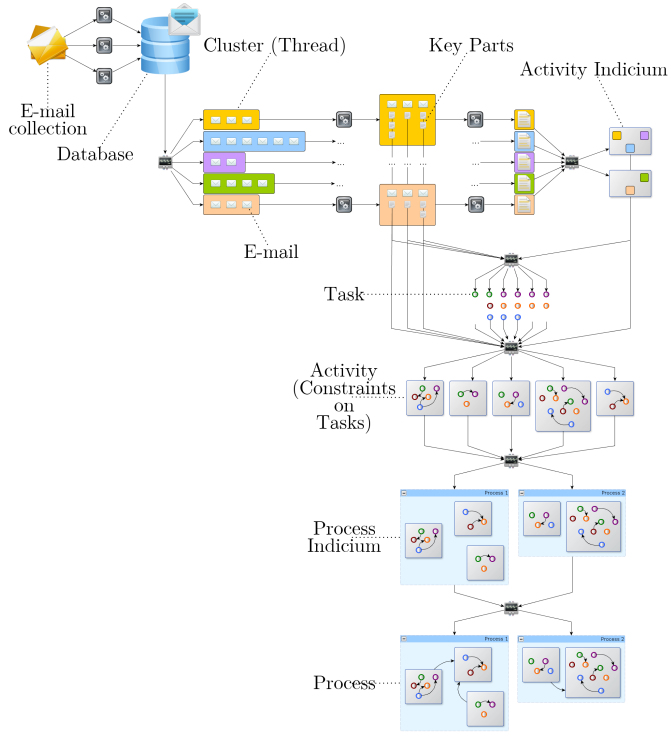


Fig. 1. The MAILOFMINE approach

properties over semi-infinite runs instead. On the contrary, human processes have an end, other than a starting point. We envision the process schemes like grammars describing the language spoken by collaborative organisms in terms of activities, thus being more related to formal languages rather than temporal logic.

The remainder of this paper is organized as follows: Section II outlines the overall approach of MAILOFMINE, then Section III describes the process model we adopt in our mining approach. Section IV describes in detail the mining technique. Section VIII presents an extensive validation of the technique, and finally Section IX draws some concluding remarks, outlining future activities.

II. THE MAILOFMINE APPROACH

The MAILOFMINE approach (and the tool we are currently developing) adopts a modular architecture, the components of which allow to incrementally refine the mining process. We describe it into three parts: *(i)* the preliminary steps, from the retrieval of e-mail messages to the reconstruction of communication threads; *(ii)* the extraction of key parts, the activities and tasks indicia detection, and the tasks definition; *(iii)* the final steps, from the activities definition to the final mined process extraction.

A. Preliminary Steps and E-mail Clustering

Initially we need to extract e-mail messages from the given archive(s). Archives are compliant to different standards, according to the e-mail client in use, hence the component for reading the messages is intended to be plug-in based: each plug-in is a component able to access different archive formats (e.g., `.pst`, Mozilla Thunderbird files, etc.) and/or online providers (e.g., Gmail accounts, etc.). The outcome is the population of a database, on the basis of which all the subsequent steps are carried out. The first of them is the clustering of retrieved messages into extended communication threads, i.e., flows of messages which are related to each other. The considered technique, detailed in [6], which is used to guess such a connection, is based not only on the Subject field (e.g., looking at "Fwd:" or "Re:" prefixes) or SMTP Header fields (e.g., reading the "In-Reply-To" field), but on the application of a more complex object matching decision method. Such indicators, indeed, though likely trustworthy, might be misleading: *(i)* in the everyday life, it is a common attitude to reply to old messages for talking about new topics, which may have nothing to do with the previous one; *(ii)* conversely, it may happen that a new message is related to a previous, though written and sent as if it were a new one.

Once the communication threads are recognized, we can assume them all as activity indicia candidates.

After the clustering into threads, messages are analyzed in order to identify key parts. Key parts are identified by adding, to a well-known technique for the removal of quoted material [26], an iterative approach over the e-mail messages in the thread. Such an approach is based on the Unix `diff` command, performed between a given e-mail and the key parts identified so far.

B. Identifying Activities

Once key parts and messages in threads are gathered, MAILOFMINE can build activity indicia as the concatenation of all the key parts. Then, the clustering algorithm is used again, this time to identify the matches between activity indicia. E.g., let us suppose to have *(i)* a thread T_{del41} related to the writing of Deliverable 4.1 of a research project, *(ii)* another thread T_{del52} related to the writing of Deliverable 5.2 of the same research project, and, finally, *(iii)* a T_{air} dealing with an airplane flight booking, for the next review meeting. Thus, the algorithm is expected to cluster as follows: $\{T_{del41}, T_{del52}\}, \{T_{air}\}$. Its output represents the activities set.

By taking into account the activities set and the key parts (task indicia candidates), the clustering algorithm checks for matching key parts, identifying them as tasks. E.g., let us suppose to have *(i)* a key part $k_{airData}$ specifying the booked airplane data, *(ii)* another key part $k_{airBook}$ containing the confirmation of the booking, and, finally, *(iii)* a k_{del41}: "Please find attached Deliverable 4.1". Thus, the algorithm is expected to cluster as follows: $\{k_{airData}, k_{airBook}\}, \{k_{del41}\}$.

Hence, each set is a task indicium for one task. The analysis, though, does not terminate here. In order to understand whether the task under investigation is clarifying or productive, MAILOFMINE checks *(i)* if any document oriented outcome in the original e-mail messages was attached; *(ii)* if key parts contain Speech Acts [27], according to the technique proposed by [28]. If at least one of the listed conditions holds, it is productive. Otherwise, it is considered as clarifying. The detection of Speech Acts is supposed to be assisted by the experts, who are required to provide a dictionary of keywords of their domain field, as in [28]. For further information on the relation between Speech Acts and workflows, see [29].

Taking as input all of the preceding outcomes, MAILOFMINE starts searching for execution constraints between tasks within the activities they belong to. For this step, we adopt the MINERful algorithm, explained further in Section IV. Specifically, on the basis of all the possible constraints, a selection of those that are valid over most of the activity indicia is made. The selected constraints for each activity are its $\mathcal{PDS}$.

C. Mining the Process

Once activities and tasks are recognized, a supervised learning process takes place, in order to cluster activities into processes. This step cannot be fully performed by the system, since no linguistic connection could exist among related activities. E.g., the activity of drawing slides and the reservation of the airplane could sound completely apart, though those slides could be the material to present

at a review meeting, hold in the city which the airplane was booked to reach.

The activity grammar should not be exposed to the user building the test and verification set, of course. Instead, the threads are shown as witnesses for the activity; hence, the user associates part of them to different processes, and the learner associates all the related activities to processes.

Once processes are identified, MAILOFMINE performs the second step for the construction of the $\mathcal{PDG}$, i.e., the mining of production rules among activities inside the same process, by using the MINERful techniques for regular pattern mining again.

III. THE PROCESS MODEL

As previously introduced, a *process scheme* (or *process* for short) is 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 or tasks. 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.

Here, we adopt the Declare [18] taxonomy of constraints as the basic language for defining artful processes in a declarative way. But whereas in Declare constraints are translated into LTL formulas, we express each constraint through regular expressions, as shown in Table I.

The $Existence(m, a)$ constraint imposes the a character to appear at least m times in the string. The $Absence(n, a)$ constraint holds if a occurs at most $n - 1$ times in the string. $Init(a)$ makes each string start with a. $RespondedExistence(a, b)$ holds if, whenever a is read, b was already read or is going to be read (i.e., no matter if before or afterwards). Instead, $Reponse(a, b)$ enforces it by forcing a b to appear after a, if a was read. $Precedence(a, b)$ forces b to occur after a as well, but the condition to be verified is that b was read - namely, you can not have any b if you did not read an a before. $AlternateResponse(a, b)$ and $AlternatePrecedence(a, b)$ both strengthen respectively $Response(a, b)$ and $Precedence(a, b)$ by stating that *each* a (b) must be followed (preceded) by at least one occurrence of b (a). The "alternation" is in that you can not have two a (b) in a row before b (a). $ChainResponse(a, b)$ and $ChainPrecedence(a, b)$, in turn, specialize $AlternateResponse(a, b)$ and $AlternatePrecedence(a, b)$, both declaring that no other character can occur between a and b. The difference between the two is in that the former is verified for each occurrence of a, the latter for each occurrence of b. $CoExistence(a, b)$ holds if both $RespondedExistence(a, b)$ and $RespondedExistence(b, a)$ hold. $Succession(a, b)$ is valid if $Response(a, b)$ and $Precedence(a, b)$ are verified. The same holds with $AlternateSuccession(a, b)$, equivalent to the conjunction of $AlternateResponse(a, b)$ and $AlternatePrecedence(a, b)$, and with $ChainSuccession(a, b)$, with respect to $ChainResponse(a, b)$ and $ChainPrecedence(a, b)$. $NotChainSuccession(a, b)$ expresses the impossibility for b to occur immediately after a. $NotSuccession(a, b)$ generalizes the previous by imposing that, if a is read, no other b can be read until the end of the string. $NotCoExistence(a, b)$ is even more restrictive: if a appears, not any b can be in the same string.

The translation of constraints into regular expressions of Table I could be optimized further, although some redundancies are kept in order to provide a better readability and give confidence to the reader in finding similarities between constraints. E.g., you can easily see how $Response$, $AlternateResponse$ and $ChainResponse$ strengthen along the hierarchy, similarly to $Precedence$, $AlternatePrecedence$ and $ChainPrecedence$. Examples are made so to give a hint on the sense of such constraints. Some characters are strongly emphasized to put in evidence how the constraint work and what it checks. The underlined characters are the "implying", namely, their presence is the condition that triggers the constraint: if they did not appear, the constraint would have had no effect on the structure of the string.

In addition to the above constraints, we consider also the ones described in Table II. Taking inspiration from the relational data model cardinality constraints, we call *(i)* $Participation(a)$ the $Existence(m, a)$ constraint for $m = 1$ and *(ii)* $Unique(a)$ the $Absence(n, a)$ constraint for $n = 2$, since the former states that a must appear at least once in the string, whereas the latter causes a to occur no more than once. $End(a)$ is the dual of $Init(a)$, in the sense that it constrain each string to end with a. Beware that $End(a)$ would be clueless in a LTL interpretation, since LTL is thought to express temporal logic formulae over infinite traces. On the contrary, it makes perfectly sense to have a concluding task for a finite process, expressed by means of a regular automaton like the one underlying any regular expression.

A. An Example

Here we outline a brief example (cf. [6]). Let us suppose to have an e-mail archive, containing various process instances indicia, and to focus specifically on the planning of a new meeting for a research project. We suppose to execute the overall MAILOFMINE technique, and we report the possible result, starting from the list of tasks in activities (Process Description 1). Then we consider the tasks of the "Agenda" activity only.

Process Description 1 Activities and tasks list

Activity:	$\langle Agenda \rangle$
Task:	p ("*p*roposeAgenda"): Productive
Actors:	{You: Contributor, $Community$: Spectator}
Duration:	4 dd.
Task:	r ("requestAgenda"): Clarifying
Actors:	{$Participant$: Contributor, $Community$: Spectator}
Duration:	$\perp$
Task:	c ("commentAgenda"): Clarifying
Actors:	{$Participant$: Contributor, $Community$: Spectator}
Duration:	$\perp$
Task:	n ("co*n*firmAgenda"): Productive
Actors:	{You: Contributor, $Community$: Spectator}
Duration:	2 dd.

We suppose that a final agenda will be committed ("confirmAgenda" – n) after that requests for a new proposal ("requestAgenda" – r), proposals themselves ("proposeAgenda" – p) and comments ("commentAgenda" – c) have been circulated.

Some terms used in the example, e.g., You, Contributor, Community, Productive, Clarifying, etc. refer to the classification MAILOFMINE operate on messages during the mining phases, and are explained in [6]. For the purposes of this paper, the reader can skip them. The aforementioned tasks and activities are bound to the following constraints. We start with the *existence* constraints of Process Description 2, each focusing on a single task.

Process Description 2 Existence constraints on the example tasks and activities

Activity:	$\langle Agenda \rangle$		
Task:	"*p*roposeAgenda",	p:	$[0, *]$
Task:	"*r*equestAgenda",	r:	$[0, *]$
Task:	"commentAgenda",	c:	$[0, *]$
Task:	"co*n*firmAgenda",	n:	$[1, 1]$, End(n)

In Process Description 3 we report the *relation* constraints, namely holding between couple of tasks.

Constraint	Regular expression	Example
Existence constraints		
$Existence(m, a)$	`[^a]*(a[^a]*){m,}[^a]*`	bc**a**a**c** for $m = 2$
$Absence(n, a)$	`[^a]*(a[^a]*){0,n}[^a]*`	bc**a**a**c** for $n = 3$
$Init(a)$	`a.*`	**a**ccbbbaba
Relation constraints		
$RespondedExistence(a, b)$	`[^a]*((a.*b)|(b.*a))*[^a]*`	**bca**accbbbaba
$Response(a, b)$	`[^a]*(a.*b)*[^a]*`	bca**a**acc**b**bb**ab**
$AlternateResponse(a, b)$	`[^a]*(a[^a]*b)*[^a]*`	bca**a**cc**b**bb**ab**
$ChainResponse(a, b)$	`[^a]*(ab[^a^b]*)*[^a]*`	bc**ab**bb**ab**
$Precedence(a, b)$	`[^b]*(a.*b)*[^b]*`	c**a**acc**b**bb**a**ba
$AlternatePrecedence(a, b)$	`[^b]*(a[^b]*b)*[^b]*`	c**a**acc**bab**a
$ChainPrecedence(a, b)$	`[^b]*(ab[^a^b]*)*[^b]*`	c**abab**a
$CoExistence(a, b)$	`[^a^b]*((a.*b)|(b.*a))*[^a^b]*`	**bca**ccbbbaba
$Succession(a, b)$	`[^a^b]*(a.*b)*[^a^b]*`	c**a**acc**b**bbab
$AlternateSuccession(a, b)$	`[^a^b]*(a[^a^b]*b)*[^a^b]*`	c**a**cc**bab**
$ChainSuccession(a, b)$	`[^a^b]*(ab[^a^b]*)*[^a^b]*`	c**abab**
Negative relation constraints		
$NotChainSuccession(a, b)$	`[^a]*(a[^a^b])*[^a]*`	bca**a**cc**b**bbb**a**
$NotSuccession(a, b)$	`[^a]*(a[^b]*)*[^a^b]*`	bc**a**a**c**c**a**
$NotCoExistence(a, b)$	`[^a^b]*((a[^b]*)|(b[^a]*))?`	c**a**a**c**c**a**

TABLE I

SEMANTICS OF DECLARE CONSTRAINTS AS REGULAR EXPRESSIONS

Constraint	Regular expression	Example
Existence constraints		
$Participation(a) \equiv Existence(1, a)$	`[^a]*(a[^a]*)+[^a]*`	bc**a**a**c**
$Unique(a) \equiv Absence(2, a)$	`[^a]*(a)?[^a]*`	bc**a**c
$End(a)$	`.*a`	bcaaccbbbab**a**

TABLE II

ADDITIONAL CONSTRAINTS IN MAILOFMINE

Process Description 3 Relation constraints on the example tasks and activities

Activity: ⟨*Agenda*⟩
$response$(r, p)
$respondedExistence$(c, p)
$succession$(p, n)

IV. THE MINERFUL ALGORITHM

MINERful is the algorithm for mining declarative constraints out of activities traces. As described in Section III, tasks can be abstracted like characters appearing over finite strings, which, in turn, represent process traces. Thus, MINERful works on collections of finite strings, actually. Therefore, it solves the more general problem of finding a specific set of regular patterns out of a number of strings. We will interchangeably use the terms "task" and "character", as well as "trace" and "string", then. MINERful is based on the concept of *MINERfulKB*: it holds all of the useful information related to a local scope in the analysis of a single activity, extracted from the given traces and tailored to the further discovery of constraints that possibly lay behind. The first step of MINERful is to synthesize such a matrix (Section VI-A), indeed, in order to easily mine the declarative model afterwards (Section VI-B).

V. DEFINITIONS

For the definition of MINERful interplay, we have to keep in mind that it is referred to a couple of characters: one is considered as the *pivot*, ρ, the other is the *searched*, σ. 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 $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

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

- $\delta(\cdot)$ $\delta : \mathbb{Z}^{\infty} \rightarrow \mathbb{N}^1$ is the *distances* function, mapping a distance between ρ and σ to the number of cases they appeared at that distance in a string (e.g., $\delta(2) = 10$ means that we have the evidence of a searched σ appearing 2 characters after the pivot ρ, as in the substring cacbcc, over 10 cases);
- $b^{\rightarrow}$ $b^{\rightarrow} \subset \mathbb{N}$ is the *in-between onwards appearances* counter (e.g., if it is equal to 2, it means that the pivot ρ appeared 2 times between the preceding occurrence of ρ and the following first occurrence of the searched σ, as in the substring accaacb);
- $b^{\leftarrow}$ $b^{\leftarrow} \subset \mathbb{N}$ is the *in-between backwards appearances* counter (e.g., if it is equal to 3, it means that the pivot ρ appeared 3 times between the following occurrence of ρ and the preceding first occurrence of the searched σ, as in the substring bcacaaca);

The *distance* represents the number of characters between ρ and σ. It is a positive value if σ *follows* ρ, negative if it *preceds* ρ. With a slight abuse of notation, we consider $\delta(+\infty)$ and $\delta(-\infty)$ 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. $\delta(0)$ counts the number of cases in which the searched σ did not appear nor before neither after ρ in the string.

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

- $\gamma(\cdot)$ $\gamma : \mathbb{N} \rightarrow \mathbb{N}^1$ is the *global appearances* function (e.g., $g(4) = 2$ means that it happened to the pivot ρ to appear exactly four times in two strings only, as for a, having aabbabccaaabab and babacaa in the analyzed collection of traces);
- g^i $g^i \in \mathbb{N}$ is the number of strings where the pivot ρ appeared as the *initial* one (e.g., if $g^i = 5$, five strings started with a);
- g^l $g^l \in \mathbb{N}$ is the number of strings where the pivot ρ appeared as the *last* one (e.g., if $g^l = 0$, no string ended with a);

Provided the information model, we describe the functions that lead from tasks to the related information.

Definition 3 (MINERful interplay function): A *total* function $f^{\mathcal{I}} : \Sigma \times \Sigma \rightarrow \mathcal{D}$, linking each couple of characters $\rho, \sigma \in \Sigma$ to a single tuple $d \in \mathcal{D}$.

Definition 4 (MINERful ownplay function): A *total* function $f^{\mathcal{S}} : \Sigma \rightarrow \mathcal{S}$, linking each character $\rho \in \Sigma$ to a single tuple $e \in \mathcal{E}$.

Given this all, we say that the MINERfulKB is an interpretation for both the MINERful interplay function and the MINERful ownplay function, namely $\mathcal{K} = \{f^{\mathcal{D}}, f^{\mathcal{E}}\}$.

Such an interpretation is given by the application of the MINERful to a collection of strings, as described in the following Section VI.

Given two characters, namely $\langle \rho, \sigma \rangle \in \Sigma_T$, where Σ_T is the actual alphabet, appearing over a set of traces T, we denote the application on ρ, σ of the MINERful interplay function, intepreted over T, as $f_T^{\mathcal{D}}(\rho, \sigma) = \mathcal{D}_{\rho,\sigma} = \langle \delta_{\rho,\sigma}, b^{\rightarrow}_{\rho,\sigma}, b^{\leftarrow}_{\rho,\sigma} \rangle$. Similarly, the application on ρ of the MINERful ownplay function, intepreted over T, is denoted as $f_T^{\mathcal{E}}(\rho) = \mathcal{E}_\rho = \langle \gamma_\rho, g^i_\rho, g^l_\rho \rangle$

We remark here that the MINERful interplay expresses a local view on two characters, measuring the distances and the alternations between the first and the second. Thus, in order to correctly interpret the MINERful interplay function, you should focus on substrings, starting from the pivot ρ and ending in the searched σ (or viceversa, in case of negative distances). Beware that such substrings are not related to the *first* occurrence of ρ in the string: *any* ρ is the initial (final) character for a following (preceding) substring ending in (starting from) σ, each separatedly analyzed. On the contrary, in order to give a correct interpretation to the MINERful ownplay function, you should take into account the whole string at once. For instance, suppose to have a string like this: aabbac. Then, we might have $\mathcal{D}_{\mathsf{a},\cdot}$ described in Table III, whereas

$$\mathcal{E}_{\mathsf{a}} = \langle \gamma_{\mathsf{a}} = \left\{ \begin{array}{ll} \langle 3, 1 \rangle & \\ \langle x, 0 \rangle & \forall x \in \mathbb{N} \setminus \{3\} \end{array} \right\}, g^i_{\mathsf{a}} = 1, g^l_{\mathsf{a}} = 0 \rangle$$

.

VI. The algorithm

Algorithm 1 The MINERful pseudo-code algorithm (bird-eye watching)

$\mathcal{K}_T \leftarrow \mathsf{computeKBOnwards}(T, \Sigma_T)$
$\mathcal{K}_T \leftarrow \mathsf{computeKBBackwards}(T, \Sigma_T)$
$\mathcal{B} \leftarrow \mathsf{discoverConstraints}(\mathcal{K}_T, \Sigma_T, |T|)$

A. Construction of the MINERfulKB

The input of this algorithm is a bag of strings, called T, and an alphabet Σ_T. The assignment of variables simply consists in the definition of the alphabet Σ_T. At the end of the run, we have the interpretations for both MINERful interplay function and MINERful ownplay function, respectively $\{f_T^{\mathcal{D}}, f_T^{\mathcal{E}}\} \equiv \mathcal{K}_T$, computed on the basis of T.

The MINERfulKB is designed 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 bag of strings). At the same time, it must be fast itself: moving the whole complexity to this step would take no advantage. The algorithm presented here is built to be completely on-line, i.e., it improves the MINERfulKB as new strings occur and as new characters in the string are read, with no need to go back on previous data in the end.

a) The algorithm: Before starting the description of the code in Algorithm, we resume here the notation adopted. *Sets* differ from *lists* in that they can not have multiple copies of the same value. Therefore, if, e.g., $X = \{x\} \implies X \cup \{x\} = \{x\}$, i.e., unions are implicitely meant to be distinct: the reader has to keep this in mind when looking at instructions like $R \leftarrow R \cup \{\sigma\}$ (see line 12 in Algorithm 2). Lists, though, have an explicit positional indexing over the values inserted. Hence, $\vec{\pi_\rho}[j]$ (see line 26 in Algorithm 2), is pointing at the j-th element in the $\vec{\pi_\rho}$ list. Strings are considered as lists of characters: thus, $t[i]$ refers to the i-th character in the string t (see line 11 in Algorithm 2). Lists and strings are provided with a

Algorithm 2 The computeKBOnwards function pseudo-code algorithm

1: $\forall x \in \mathbb{Z}^{\infty}.\ \delta(x) := 0$
2: $\forall x \in \mathbb{N}.\ \gamma(x) := 0$
3: **for all** $t \subseteq T$ **do**
4: $\quad g^i_{t[0]} := g^i_{t[0]} + 1$
5: $\quad R \leftarrow \emptyset$ # R: set of characters already appeared in t
6: $\quad \forall r, s \in \Sigma_T\ N_{r,s} \leftarrow 0$ # N: counter for missing s characters after r
7: $\quad \forall r \in \Sigma_T,\ \vec{\pi_r} \leftarrow \{\}$ # $\vec{\pi_r}$: vector of indexes where r appears in t
8: $\quad \forall r, s \in \Sigma_T,\ W_{r,s} \leftarrow 0$ # W: counter for r's repeated before the next s
9: $\quad \forall r, s \in \Sigma_T,\ \widehat{W_{r,s}} \leftarrow \bot$ # $\widehat{W}$: flag for granting the update of W
10: $\quad$ **for** $i = 1$ **to** $|t|$ **do**
11: $\quad\quad \sigma \leftarrow t[i]$
12: $\quad\quad R \leftarrow R \cup \{\sigma\}$
13: $\quad\quad \vec{\pi_\sigma} \leftarrow \vec{\pi_\sigma} \circ \{i\}$
14: $\quad\quad$ **for all** $\rho \in R$ **do**
15: $\quad\quad\quad$ **if** $\rho = \sigma$ **then**
16: $\quad\quad\quad\quad$ **for all** $s \in \Sigma_T \setminus \{\rho\}$ **do**
17: $\quad\quad\quad\quad\quad N_{\rho,s} \leftarrow N_{\rho,s} + 1$
18: $\quad\quad\quad\quad\quad$ **if** $\widehat{W_{\rho,s}} = \bot$ **then**
19: $\quad\quad\quad\quad\quad\quad \widehat{W_{\rho,s}} \leftarrow \top$
20: $\quad\quad\quad\quad\quad$ **else**
21: $\quad\quad\quad\quad\quad\quad W_{\rho,s} \leftarrow W_{\rho,s} + 1$
22: $\quad\quad\quad\quad\quad$ **end if**
23: $\quad\quad\quad\quad$ **end for**
24: $\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad$ **for** $j = 1$ **to** $|\vec{\pi_\rho}|$ **do**
26: $\quad\quad\quad\quad\quad \delta_{\rho,\sigma}(i - \vec{\pi_\rho}[j]) := \delta_{\rho,\sigma}(i - \vec{\pi_\rho}[j]) + 1$
27: $\quad\quad\quad\quad$ **end for**
28: $\quad\quad\quad\quad N_{\rho,\sigma} \leftarrow 0$
29: $\quad\quad\quad\quad$ **if** $\widehat{W_{\rho,\sigma}} = \top$ **then**
30: $\quad\quad\quad\quad\quad b^{\rightarrow}_{\rho,\sigma} := b^{\rightarrow}_{\rho,\sigma} + W_{\rho,\sigma}$
31: $\quad\quad\quad\quad\quad \widehat{W_{\rho,\sigma}} \leftarrow \bot,\ W_{\rho,\sigma} \leftarrow 0$
32: $\quad\quad\quad\quad$ **end if**
33: $\quad\quad\quad$ **end if**
34: $\quad\quad$ **end for**
35: $\quad$ **end for**
36: $\quad$ **for all** $r \in R$ **do**
37: $\quad\quad$ **for all** $\bar{s} \in \Sigma_T \setminus R$ **do**
38: $\quad\quad\quad \delta_{r,\bar{s}}(0) := \delta_{r,\bar{s}}(0) + |\vec{\pi_{\bar{s}}}|$
39: $\quad\quad$ **end for**
40: $\quad\quad$ **for all** $\bar{s} \in N_r$ **do**
41: $\quad\quad\quad \delta_{r,\bar{s}}(+\infty) := \delta_{r,\bar{s}}(+\infty) + N_{r,s}$
42: $\quad\quad$ **end for**
43: $\quad\quad$ **if** $|\vec{\pi_r}| = 1$ **then**
44: $\quad\quad\quad \delta_{r,r}(+\infty) := \delta_{r,r}(+\infty) + 1$
45: $\quad\quad$ **end if**
46: $\quad$ **end for**
47: $\quad$ **for all** $s \in \Sigma_T$ **do**
48: $\quad\quad \gamma_s(|\vec{\pi_s}|) := \gamma_s(|\vec{\pi_s}|) + 1$
49: $\quad$ **end for**
50: $\quad g^l_{t[|t|]} := g^l_{t[|t|]} + 1$
51: **end for**

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

$b^{\rightarrow}_{\mathsf{a,a}} = 0$; $b^{\leftarrow}_{\mathsf{a,a}} = 0$
$b^{\rightarrow}_{\mathsf{a,b}} = 1$; $b^{\leftarrow}_{\mathsf{a,b}} = 0$
$b^{\rightarrow}_{\mathsf{a,c}} = 2$; $b^{\leftarrow}_{\mathsf{a,c}} = 0$

TABLE III
EXAMPLES OF STATISTICAL VALUES STORED IN THE INTERPRETED MINERFULKB

concatenation function, $\circ$: for instance, the effect of $\vec{\pi_\sigma} \leftarrow \vec{\pi_\sigma} \circ \{i\}$ is to add i as the last element in $\vec{\pi_\sigma}$ (see line 13 in Algorithm 2). For pointing at a specific element in a map (indexed multi-set), we put the "coordinates" in the subscript of the set identificator: e.g., $N_{r,s}$ is the element in N corresponding to r and s (see line 6 in Algorithm 2).

In order to ease the reader to distinguish between assignments of temporary variables (like the ones from line 5 to line 9 in Algorithm 2) and the update of the interpretation for the MINERfulKB (see e.g., line 4 in Algorithm 2), we denote the former with $\leftarrow$, the latter with $:=$.

We consider passes by reference in the invocation of procedures, so that passed objects are subjected to direct side effects from within the procedure body.

b) Explanation of Algorithm 2: From line 1 to line 2, the interpretations of the γ and δ functions are initialized, supposing that they are constant and equal to 0, whatever the value the variables assume. Then, for each string t in T (line 3), the first character appearing ($t[0]$) is recorded into the related $g^i_{t[0]}$ as the first (line 4). After the initialization of auxiliary data structures, whose role is briefly explained in-line on the code itself and further in this Section, the analysis of the single characters in the string begins (line 10). First of all, the encountered character σ is added to the set of appeared characters in t, namely R (if it is not already in – see VI-A.0.b). Next, the current index is concatenated ($\circ$ operation) to the list of positions where σ appeared in t ($\vec{\pi_\sigma}$), at line 13. On line 14 the algorithm starts the computation of interleaving statistics between characters.

For each of the characters already appeared in the string, ρ, the algorithm proceeds differently, depending on whether a character already appeared is read again ($\rho = \sigma$) or not (line 15).

In the first case, the temporary counter for cases in which s (where s is any other character in Σ_T but ρ) did not appear anymore after an occurrence of ρ ($N_{\rho,s}$) is incremented by 1 (line 17). This is due to the fact that such counter will be reset if s appears afterwards (see line 28), and its value is going to be "flushed" to $\delta_{r,\bar{s}}(+\infty)$ at the end of the string t (see line 41). From line 18 to line 22, the algorithm updates the counters for repeated occurrences of ρ before the next occurrence of s: $\widehat{W_{\rho,s}}$ is the flag for incrementing the $W_{\rho,s}$ counter; hence, if it is set to false, it gets true, whereas if it is already true, $W_{\rho,s}$ is incremented by one. This is due to the fact that when the next occurrence of s is found in the string, the value of $W_{\rho,s}$ will be flushed as an increment to $b^{\rightarrow}_{\rho,s}$ (see line 30), before $\widehat{W_{\rho,s}}$ and $W_{\rho,s}$ are reset, respectively, to $\perp$ and 0, (line 31).

If the encountered σ differs from ρ in the loop over R, then the value assumed by $\delta_{\rho,\sigma}$ at the current distance between ρ and σ has to be incremented by 1. Though, we may have not only one position where ρ appeared, but many. Think to aaccccacab..., for instance: there, the pivot a appeared at position 1, 2, 7 and 9, and the searched b at position 10. Thus, b must be recorded to appear at distance 1, 3, 8 and 9 from a. Reminding that $\vec{\pi_\rho}$ collects all of the indexes where ρ is read (see line 13), this is what happens at line 26, actually – repeated for each position of ρ in $\vec{\pi_\rho}$, i.e., inside the loop starting at line 25. This is probably one of the most difficult steps of the algorithm, though it prevents the analysis to be repeated like a transitive closure on each string for each appeared character to gather information. As we said at the beginning of Section VI-A, the analysis for the MINERful interplay to get interpreted must be local to each occurrence of ρ, but the construction of the MINERfulKB had not to be too complex: this is the most noticeable example of how we managed both the requirements.

The final part of the outermost cycle updates counters on the basis of the previously gathered information. The instruction of line 38 records the number of times that the read character r occurred in t as the counter of how many times s (which was not read) missed, i.e., once for each of the appeared r. The statement at line 44 is due to the need of recording that if r appeared once and then no more in the string, then $\delta_{r,r}(+\infty)$ must be incremented by one in the interpretation. This is the only case where this operation makes sense. If we had used for $\delta_{r,r}(+\infty)$ the technique used at line 38, it would have been meaningless, since always occurs a last r, after which no more r are read afterwards (this is the reason why the cycle for computing the value of $N_{r,s}$ is executed for each $s \neq r$ – see line 16). On line 48, the function distributing the number of appearances of each character s in the alphabet Σ_T over T is updated: the number of occurrences of s in t, namely $|\vec{\pi_s}|$, is the argument, and the referred value is incremented by 1. $|\vec{\pi_s}|$ can be 0 as well, if it was never read in t. In the end (line 50), the counter for the appearances as last for the ending character of $\vec{t}$ is incremented by 1.

c) A running example for the computation of the δ function and the b counter: Since the work of the algorithm on δ and b (thus respectively on N, and on W and $\widehat{W}$) can lead to some difficulties in the understanding, we explain it through an example. Suppose to have a t string like this: aabbac. Taking into account the analysis of a only as the pivot ρ, for sake of simplicity, the evolution of N_{a} throughout the algorithm is reported on the following Table IV, as W_{a} and $\widehat{W_{\mathsf{a}}}$ evolve like on Table V.

d) On the computeKBBackwards procedure: This algorithm is called twice, one reading strings onwards (computeKBOnwards), i.e. from left to the right (according to the Western Latin standard), one backwards (computeKBBackwards). Despite this, here we reported the pseudo-algorithm of the former only, since the latter is almost identical. The only differences are in that computeKBBackwards:

- it does not update either the γ function, nor the g^i nor the g^l (namely, it does not contribute to give an interpretation to the MINERful ownplay, being this done by computeKBOnwards already);
- it does not update the δ function for 0 values (since computeKBOnwards already detected characters never appeared in the string, if any);
- it reverses the sign of i, the counter of the current index in the string (namely, it is initialized with -1 and proceeds being decremented by 1 at each step);
- it updates the δ function for $-\infty$ values, instead of $+\infty$, whenever the same conditions of lines 40 and 43 in Algorithm 2 are verified.

e) Discussion on the complexity: In the following, we discuss the complexity of Algorithm 2.

$\langle N_{\mathsf{a}}, \delta_{\mathsf{a},\cdot}\rangle \backslash^{\sigma \in t}$	a	a	b	b	a	c	
$\langle N_{\mathsf{a,b}}, \delta_{\mathsf{a,b}}\rangle$	$\langle 1, -\rangle$	$\langle 2, -\rangle$	$\langle 0, -\rangle$	$\langle 0, -\rangle$	$\langle 1, -\rangle$	$\langle 1, -\rangle$	$\langle 0, +1\rangle$
$\langle N_{\mathsf{a,c}}, \delta_{\mathsf{a,c}}\rangle$	$\langle 1, -\rangle$	$\langle 2, -\rangle$	$\langle 3, -\rangle$	$\langle 4, -\rangle$	$\langle 5, -\rangle$	$\langle 0, -\rangle$	$\langle 0, +0\rangle$

TABLE IV
THE EVOLUTION OF N_{A} AND $\delta_{\mathsf{A,B}}$, $\delta_{\mathsf{A,C}}$, OVER THE READING OF A STRING t

$\langle\langle \widehat{W_{\mathsf{a}}}, W_{\mathsf{a}}\rangle, b^{\rightarrow}_{\mathsf{a},\cdot}\rangle \backslash^{\sigma \in t}$	a	a	b	b	a	c
$\langle\langle \widehat{W_{\mathsf{a,b}}}, W_{\mathsf{a,b}}\rangle, b^{\rightarrow}_{\mathsf{a,b}}\rangle$	$\langle\langle \top, 0\rangle, -\rangle$	$\langle\langle \top, 1\rangle, -\rangle$	$\langle\langle \bot, 0\rangle, +1\rangle$	$\langle\langle \bot, 0\rangle, -\rangle$	$\langle\langle \top, 0\rangle, -\rangle$	$\langle\langle \top, 0\rangle, -\rangle$
$\langle\langle \widehat{W_{\mathsf{a,c}}}, W_{\mathsf{a,c}}\rangle, b^{\rightarrow}_{\mathsf{a,c}}\rangle$	$\langle\langle \top, 0\rangle, -\rangle$	$\langle\langle \top, 1\rangle, -\rangle$	$\langle\langle \top, 1\rangle, -\rangle$	$\langle\langle \top, 1\rangle, -\rangle$	$\langle\langle \top, 2\rangle, -\rangle$	$\langle\langle \bot, 0\rangle, +2\rangle$

TABLE V
THE EVOLUTION OF $\widehat{W_{\mathsf{A}}}$, W_{A}, AND $b^{\rightarrow}_{\mathsf{A}}$ OVER THE READING OF A STRING t

Lemma 1: The procedure for building the knowledge base of the MINERful is *(i)* linear time w.r.t. the number of strings in the testbed, *(ii)* quardratic time w.r.t. the size of strings in the testbed, *(iii)* quadratic time w.r.t. the size of the alphabet; therefore, the complexity is $O(|T| \cdot |t_{max}|^2 \cdot |\Sigma_T|^2)$.

Proof: The outermost cycle (line 3) is repeated exactly $|T|$ times, the following inner (line 10), $|t|$ times for each $t \in T$. In the worst case, i.e., assuming that each string is as long as the longest, it loops $|t_{max}|$ time, where $t_{max} = \max t \in T|t|$. Actually, such couple of loops make the instructions be repeated exactly $\sum_{t \in T} |t|$ times, i.e., the relevant size of the input. At line 14, we have a cycle whose number of repetitions grows as new characters are found in the analyzed string. The number of loops depends on the size of the string $|t|$ and the size of the alphabet Σ_T at the same time. In fact, we might assume that each character read was not found before in the string. So, as soon as a new character is read, you have one loop more. You might say that, hence, the instructions in the block are executed $1+2+3+\cdots+|t|$ times as the cursor in the string moves on. If it was so, loops starting at line 10 and line 14) had run at most

$$\frac{|t| \times (|t|+1)}{2}$$

times, as in the formula for counting the sum of the first $|t|$ natural numbers. Although, the maximum amount of "new" characters is bounded by the characters you can really have. Therefore, assuming the worst case, i.e., all of the characters of Σ_T in every string, it runs at most $1+2+3+\cdots+|\Sigma_T|$ times. Then, we have to subtract the number of loops that are not executed due to the limitation of the alphabet size (if the alphabet size is smaller than the size of the strings, which is likely). Let:

$$\Delta_{t,\Sigma_T} = |t| - |\Sigma_T|$$

Thus, the number of loops is equal to:

$$\frac{|t| \times (|t|+1)}{2} - \frac{\Delta_{\Sigma_T,t} \times (\Delta_{\Sigma_T,t}+1)}{2} \cdot \Theta(\Delta_{t,\Sigma_T} - 1)$$

where $\Theta(x)$ is the Heaviside step function (equal to 0, and thus deleting the second term in the subtraction if Δ_{t,Σ_T}, i.e., if $|t| < |\Sigma_T|$, otherwise equal to 1). If we suppose that $|t| < |\Sigma_T|$, then we can simplify terms of the moltiplications and subtractions, up to

$$\frac{2|\Sigma_T||t| - |\Sigma_T|^2 + |\Sigma_T|}{2} = \frac{2\Delta_{t,\Sigma_T}|\Sigma_T| + |\Sigma_T|}{2} \leqslant |\Sigma_T||t|$$

Depending on the condition at line 15, the algorithm enters one of the two innermost loops, one starting at line 16, the other at line 25.

The first is executed exactly $|\Sigma_T| - 1$ times, no matter the outer cycles. The second, instead, is such that the more repetitions of the same character in the string we had, the more it loops. If we had strings composed by concatenations of the same character (the worst case for such cycle) following a prefix with all of the alphabet (the worst case for the outer cycle), this loop asymptotically causes on the long run as many loops as $|t|$.

The instructions in the bodies of the loops are readings and writings in memory[1] so they do not add any relevant degree of complexity to the algorithm.

Summing up this computation analysis, we have that the procedure algorithm is

$$O\left(\underbrace{|T|}_{\text{loop at 3}} \underbrace{|t_{max}||\Sigma_T|}_{\text{loops at 10 and 14}} \left(\underbrace{|\Sigma_T|}_{\text{loop at 16}} + \underbrace{|t_{max}|}_{\text{loop at 25}}\right)\right)$$

■

B. Discovery of Constraints

Artful processes are represented by means of a set of constraints, imposing the rules that each process instance must follow, whatever the execution trace is. The set of mined constraints are those described in Tables I and II, that are an extended version of those explained in [18]. Here, we express such constraints like predicates over the MINERfulKB, that are easily transposed into instructions for a verification algorithm.

f) Existence constraints:

$$\begin{aligned} Participation(r) &\equiv Existence(1, r) \\ &\equiv \left(\min_{\langle o,p\rangle \in \gamma_r | p>0} o > 0\right) \end{aligned} \tag{VI.1}$$

Each string had at least 1 occurrences of r in.

$$\begin{aligned} Unique(r) &\equiv Absence(2, r) \\ &\equiv \left(\max_{\langle o,p\rangle \in \gamma_r | p>0} o \leqslant 2\right) \end{aligned} \tag{VI.2}$$

There was no string with more than 1 occurrence of r in.

$$Init(r) \equiv \left(|T| \leqslant g_r^i\right) \tag{VI.3}$$

Every string starts with r.

$$End(r) \equiv \left(|T| \leqslant g_r^f\right) \tag{VI.4}$$

Every string finishes with r.

[1]The reader might ask the opportunity to analyze the complexity of searching the datum to overwrite in the temporary data structures, such as N. Although, considering that *(i)* we can exploit the alphanumeric ordering function for ordering the couples of characters, and *(ii)* the alphabet of characters is known a priori, we can easily make use of a hashing function, so that reaching the datum and overwriting it is O(1).

Rather than giving 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 $Existence(N, r)$ constraints hold, for N ranging from 0 to $\min_{\langle o,p\rangle\in\gamma_r|p>0}$ (and dually consider $Absence(N+1, r)$ valid for each N from $\max_{\langle o,p\rangle\in\gamma_r|p>0} o$ onwards), we preferred to introduce a looser couple of constraints, stating whether a task r must be executed ($Participates(r)$) or not, and whether it must be done no more than once ($Unique(r)$). 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. Finally, we want to remark the introduction of the $End(r)$ constraint: if it holds, it means that each process instance must terminate with an execution of r[2].

g) Relation constraints:

$$RespondedExistence(r, s) \equiv \neg(\delta_{r,s}(0) > 0)) \tag{VI.5}$$

There was no string such that s was not read if r was.

$$Response(r, s) \equiv RespondedExistence(r, s) \wedge \neg(\delta_{r,s}(+\infty) > 0)) \tag{VI.6}$$

There was no string such that s did not succeed r.

$$AlternateResponse(r, s) \equiv Response(r, s) \wedge b^{\rightarrow}_{r,s} = 0 \tag{VI.7}$$

There was no string such that r appeared again before the succeeding.

$$ChainResponse(r, s) \equiv AlternateResponse(r, s) \wedge \delta_{r,s}(1) \geqslant \sum_{\langle o,p\rangle\in\gamma_r} o \cdot p \tag{VI.8}$$

Each time you have an occurrence of r, the total amount of which is computed as $\sum_{\langle o,p\rangle\in\gamma_r} o \cdot p$, there is always a new s immediately following (i.e., at a distance equal to 1).

Dually, we have the following $Precedence$-based constraints.

$$Precedence(r, s) \equiv Response(r, s) \wedge \neg(\delta_{r,s}(-\infty) > 0)) \tag{VI.9}$$

There was no string such that s did not precede r.

$$AlternatePrecedence(r, s) \equiv Precedence(r, s) \wedge b^{\leftarrow}_{r,s} = 0 \tag{VI.10}$$

There was no string such that r appeared again before the preceding s.

$$ChainPrecedence(r, s) \equiv AlternateResponse(r, s) \wedge \delta_{r,s}(-1) \geqslant \sum_{\langle o,p\rangle\in G_r} o \cdot p \tag{VI.11}$$

Each time you have an occurrence of r, the total amount of which is computed as $\sum_{\langle o,p\rangle\in\gamma_r} o \cdot p$, there is always a new s immediately preceding (i.e., at a distance equal to -1).

You derive the following.

$$CoExistence(r, s) \equiv RespondedExistence(r, s) \wedge RespondedExistence(b, a) \tag{VI.12}$$

[2] $End(r)$ does not necessarily imply that as r is enacted, the process instance must get to an end, but only that the final character in the string is r.

Constraint	Symbol
Existence constraints	
$Participation(a)$	$\top^{1+}_{\rho}$
$Unique(a)$	$\top^{1-}_{\rho}$
$Init(a)$	$\top^{i}_{\rho}$
$End(a)$	$\top^{l}_{\rho}$
Relation constraints	
$RespondedExistence(a, b)$	$\top_{\rho,\sigma}$
$Response(a, b)$	$\top^{\rightarrow}_{\rho,\sigma}$
$AlternateResponse(a, b)$	$\top^{\Rightarrow}_{\rho,\sigma}$
$ChainResponse(a, b)$	$\top^{\Rrightarrow}_{\rho,\sigma}$
$Precedence(a, b)$	$\top^{\leftarrow}_{\rho,\sigma}$
$AlternatePrecedence(a, b)$	$\top^{\Leftarrow}_{\rho,\sigma}$
$ChainPrecedence(a, b)$	$\top^{\Lleftarrow}_{\rho,\sigma}$
$CoExistence(a, b)$	$\top^{\rho}_{\sigma}$
$Succession(a, b)$	$\top^{\leftrightarrow}_{\rho,\sigma}$
$AlternateSuccession(a, b)$	$\top^{\Leftrightarrow}_{\rho,\sigma}$
$ChainSuccession(a, b)$	$\top^{\Lleftrightarrow}_{\rho,\sigma}$
Negative relation constraints	
$NotCoExistence(a, b)$	$\bot^{\rho}_{\sigma}$
$NotSuccession(a, b)$	$\top^{\not\leftrightarrow}_{\rho,\sigma}$
$NotChainSuccession(a, b)$	$\top^{\not\Leftrightarrow}_{\rho,\sigma}$

TABLE VI
SYMBOLS EXPRESSING THE VALIDITY OF CONSTRAINTS

$$Succession(r, s) \equiv Response(r, s) \wedge Precedence(r, s) \tag{VI.13}$$

$$AlternateSuccession(r, s) \equiv AlternateResponse(r, s) \wedge AlternatePrecedence(r, s) \tag{VI.14}$$

$$ChainSuccession(r, s) \equiv ChainResponse(r, s) \wedge ChainPrecedence(r, s) \tag{VI.15}$$

h) Negative relation constraints:

$$NotChainSuccession(r, s) \equiv \neg(\delta_{r,s}(1) > 0)) \tag{VI.16}$$

It never happens that, after r, s follows unless you have at least another character in the middle (i.e., s never appears at distance 1 from r).

$$NotSuccession(r, s) \equiv NotChainSuccession(r, s) \wedge \sum_{\langle o,p\rangle\in G_r} o \cdot p \leqslant \delta_{r,s}(+\infty) \tag{VI.17}$$

It never happens that, after r, s follows.

$$NotCoExistence(r, s) \equiv NotSuccession(r, s) \wedge \sum_{\langle o,p\rangle\in G_r} o \cdot p \leqslant D_{r,s}(0) \tag{VI.18}$$

It never happens that, if r is in a string, s appears in the same, neither before nor aftwerwards.

i) The algorithm: In the pseudo-code of the algorithm for guessing the relation constraints (Algorithms 3, 4, 5), we assume the aforementioned predicates as the body of homonim functions. Actually, a hierarchy between constraints exists: the conjunctions inside, e.g., $ChainPrecedence(r, s)$, involving $AlternatePrecedence(r, s)$, is explicity put there in purpose. This way, the reader can have an immediate evidence of the fact that, e.g., once $ChainPrecedence(r, s)$ is known to hold, $AlternatePrecedence(r, s)$, and recursively $Precedence(r, s)$ as well, hold too.

With a slight conceptual modification w.r.t. computeKBOnwards, then, we suppose the algorithm to fill a bag of predifined constants (each symbol corresponding to the validity of a constraint for the given set of traces T), rather than giving an interpretation to each predicate (see Table VI for a reference). As the reader can see in Algorithms 3, 4, 5 and the further discussion, such constants will be added to the bag avoiding redundancies. The user who wants to understand the discovered artful process is not interested to read about trivial deductions: for instance, it is enough to say that $ChainPrecedence(r, s)$ holds, rather than explicitly return as valid constraints $ChainPrecedence(r, s)$, $AlternatePrecedence(r, s)$ and $Precedence(r, s)$ (where the latter couple is directly implied by the first). Reporting the validity of all three would add no bit of information and make the result far less readable – which is definitely to avoid, in our case, being knowledge workers the target of our approach, i.e., people with a little amount of time to dedicate to the process analysis. For the same reason, characters never appeared in the testbed are not involved neither in existence constraints nor in relation constraints related to them as implying (see line 2 in Algorithm 3).

This is the rationale underlying the nested *if* structure of the Algorithms.

j) Discussion on the complexity: In the following we discuss the complexity of the concurrent execution of Algorithms 3, 4 and 5.

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 $O(|\Sigma_T|^2)$.

Proof: We have two nested cycles, both for ρ ranging over the characters of the alphabet Σ (see lines 1 and 1 in Algorithm 3), thus both looping for $|\Sigma_T|$, at most, due to the presence of the check at line 2 in Algorithm 3), so to analyze every possible couple. The nested if statements checks whether a constraint holds or not. At most, it means that, for each couple of characters ρ and σ, you have up to 14 checks (for the innermost loop). The outermost loop calls up to 4 procedures for checking existence constraints. ■

VII. The complexity of the MINERful algorithm

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

Proof: Directly follows from Lemmata 1 and 2. ■

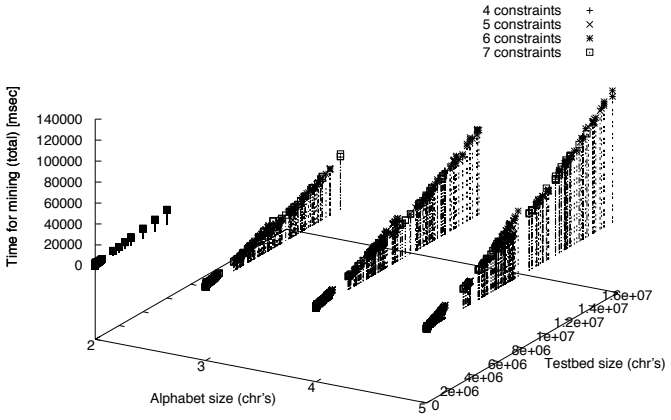
VIII. Validation and Experiments

A validation of the technique has been performed by using the example outlined in Section III-A as the starting point. Its aim is to show the efficiency of the mining technique, being the underlying algorithm polynomial wrt. the dimension of the input.

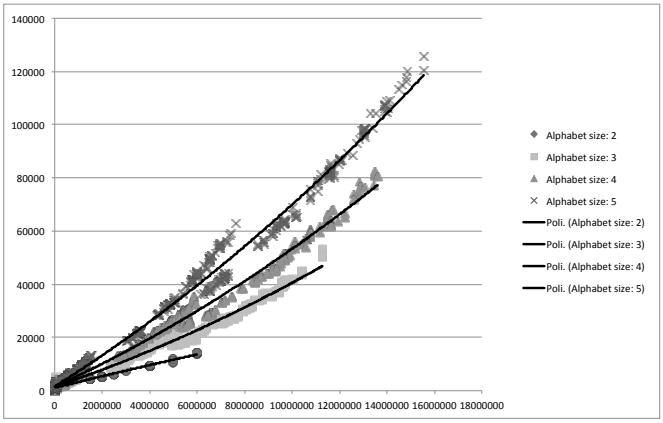
We tested the algorithm by varying the input in terms of alphabet size (different characters appearing in the strings), number of constraints valid over the strings, range of the number of characters per string. Starting with two tasks, n and p, and the related constraints, we made various experiments making the alphabet grow up to the original, thus including also r and c, plus an unconstrained task, e. The constraints ranged from the minimal set of four ($Unique(n)$, $Participation(n)$, $End(n)$, $Succession(p, n)$) to the maximal set of seven (including $Response(r, p)$, $RespondedExistence(c, p)$, $AlternatePrecedence(r, c)$). The lengths of the strings ranged through intervals of $\{[2 \dots 8], [3 \dots 12], [4 \dots 16], [5 \dots 20]\}$. The number of strings ranged as the power of 10, from 100 up to 1000000 with an exponential step. For each of the preceding combination,

Algorithm 3 The discoverConstraints function pseudo-code algorithm: non-negative relation constraints and calls to the other procedures.

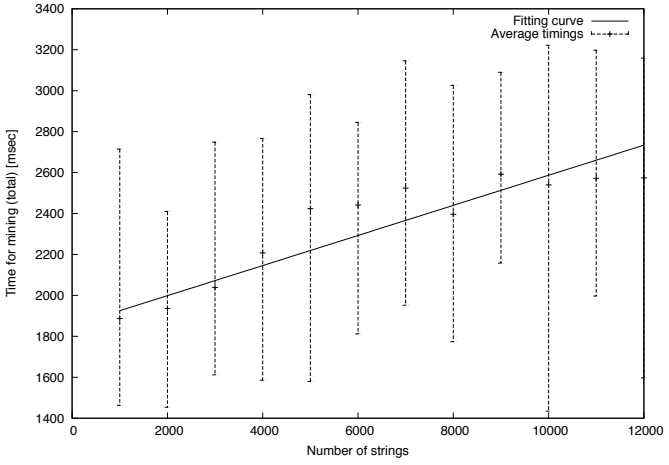
1: **for all** $\rho \in \Sigma_T$ **do**
2: **if** $\left(\max_{\langle o,p\rangle \in \gamma_r | p>0} o \leqslant 0\right)$ **then**
3: discoverExistenceConstraints(ρ, $\mathcal{K}$)
4: **for all** $\sigma \in \Sigma_T$ **do**
5: **if** $RespondedExistence(\rho, \sigma, \mathcal{K})$ **then**
6: **if** $Response(\rho, \sigma, \mathcal{K})$ **then**
7: **if** $AlternateResponse(\rho, \sigma, \mathcal{K})$ **then**
8: **if** $ChainResponse(\rho, \sigma, \mathcal{K})$ **then**
9: **if** $ChainPrecedence(\rho, \sigma, \mathcal{K})$ **then**
10: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Leftrightarrow}_{\rho,\sigma}\}$
11: **else**
12: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Rightarrow}_{\rho,\sigma}\}$
13: **end if**
14: **else**
15: **if** $AlternatePrecedence(\rho, \sigma, \mathcal{K})$ **then**
16: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Leftrightarrow}_{\rho,\sigma}\}$
17: **else**
18: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Rightarrow}_{\rho,\sigma}\}$
19: **end if**
20: **end if**
21: **else**
22: **if** $Precedence(\rho, \sigma, \mathcal{K})$ **then**
23: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\leftrightarrow}_{\rho,\sigma}\}$
24: **else**
25: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\rightarrow}_{\rho,\sigma}\}$
26: **end if**
27: **end if**
28: **end if**
29: **if** $Precedence(\rho, \sigma, \mathcal{K})$ **then**
30: **if** $AlternatePrecedence(\rho, \sigma, \mathcal{K})$ **then**
31: **if** $ChainPrecedence(\rho, \sigma, \mathcal{K}) \wedge \neg ChainResponse(\rho, \sigma, \mathcal{K}$ **then**
32: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Leftarrow}_{\rho,\sigma}\}$
33: **else**
34: **if** $\neg AlternateResponse(\rho, \sigma, \mathcal{K})$ **then**
35: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\Leftarrow}_{\rho,\sigma}\}$
36: **end if**
37: **end if**
38: **else**
39: **if** $\neg Response(\rho, \sigma, \mathcal{K})$ **then**
40: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\leftarrow}_{\rho,\sigma}\}$
41: **end if**
42: **end if**
43: **end if**
44: **if** $\neg (Response(\rho, \sigma, \mathcal{K}) \vee Precedence(\rho, \sigma, \mathcal{K}))$ **then**
45: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top_{\rho,\sigma}\}$
46: **end if**
47: **end if**
48: **if** $\top_{\rho,\sigma} \in \mathcal{B} \wedge \top_{\sigma,\rho} \in \mathcal{B}$ **then**
49: $\mathcal{B} \leftarrow \mathcal{B} \setminus \{\top_{\rho,\sigma}, \top_{\sigma,\rho}\} \cup \{\top^{\rho}_{\sigma}\}$
50: **end if**
51: guessNegativeConstraints(ρ, σ, $\mathcal{K}$)
52: **end for**
53: **end if**
54: **end for**



(a) Time needed for the execution, with respect to the testbed size and the alphabet size, setting the number of imposed constraints as the parameter



(b) Execution time (ordinates), with respect to the testbed size (abscissae), setting the alphabet size as the parameter



(c) Execution time, with respect to the number of strings, keeping fixed the size of strings, the alphabet length and the number of constraints

Fig. 2. Experimental results

Algorithm 4 The discoverNegativeConstraints procedure pseudo-code algorithm

if $NotCoExistence(\rho, \sigma, \mathcal{K})$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\bot^{\rho}_{\sigma}\}$
else
  if $NotSuccession(\rho, \sigma, \mathcal{K})$ **then**
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\nleftrightarrow}_{\rho,\sigma}\}$
  else
    if $NotChainSuccession(\rho, \sigma, \mathcal{K})$ **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{\not\Leftrightarrow}_{\rho,\sigma}\}$
    end if
  end if
end if

Algorithm 5 The discoverExistenceConstraints procedure pseudo-code algorithm

if $Participation(\rho)$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{1+}_{\rho}\}$
end if
if $Unique(\rho)$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{1-}_{\rho}\}$
end if
if $Init(\rho)$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{i}_{\rho}\}$
end if
if $End(\rho)$ **then**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{\top^{l}_{\rho}\}$
end if

10 runs were performed, for a total amount of 4480 executions. The random strings were created by Xeger[3], a Java open-source library for generating random text from regular expressions. All of the parameters, and not only constraints, were expressed in terms of regular expressions indeed, and their conjunction passed to the Xeger engine.

The machine was a Sony VAIO VGN-FE11H (an 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.

As the reader can see in Figure 2(a), the number of constraints does not affect the time taken by the algorithm to run – indeed, you are not able to distinguish between the curve designed by a group of points and another, where each group is related to a given number of constraints.

Figure 2(b) shows the fitting curves of the time taken by the algorithm to run, in comparison with the total amount of characters in input. It is a section of a parabola, confirming that the the algorithm is quadratic w.r.t. the size of the strings.

The algorithm is linear in the size of the collection of traces. In order to test this, we made a slightly different set-up: we fixed the number of constraints(7), the number of characters per string (10), and the alphabet length (5), whereas the number of strings ranged from 1000 to 12000 with a step of 1000. The result is depicted in Figure 2(c).

Furthermore, we report here by evidence that the time to build the MINERfulKB is the hardest task of the algorithm, with respect to the mining of constraints: in the worst case (1000000 strings, 7 constraints, 5 characters, length ranging from 5 to 20) the MINERfulKB was computed in 125.78 seconds whereas constraints were discovered in less than half a second (47 msec).

These results confirm Theorem 1 by experiment. There, you can

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

also see how the major impact on performances is given by the total amount of characters read, in practice. The graph showing the time needed by the algorithm to terminate, with respect to the number of characters in the testbed, fits a quadratic curve, although the shape of such parabola is very flatten, due to the nature of its non-linearity: looking back to the algorithm, it is caused by a loop cycling more as characters are repeated (for defining the $\delta_{\rho,\sigma}$ function), nested in a loop cycling more as different characters are encountered (i.e., executing the former loop for each unread ρ) – both start to grow together in terms of cycles only if strings tend to be far longer than the size of the alphabet. Finally, it is remarkable that the time needed by the discoverConstraints procedure is far shorter than what computeKBOnwards and computeKBOnwards take. Nonetheless, the latter couple is able to process a huge amount of characters in an acceptably small amount of time, as it was required to be.

Indeed, discovered constraints lead to an interesting update to the example. $Succession(p, n)$ was not reported, whereas $AlternatePrecedence(p, n)$ and $Response(p, n)$ were returned as valid. This was not a mistake of MINERful: on the contrary, as suggested by the output of the algorithm, being n the last activity, with $Unique(n)$ holding, n could not ever be repeated before p, hence not $Succession(p, n)$ but the couple of $AlternatePrecedence(p, n)$ and $Response(p, n)$ constraints explained better the structure of the process.

IX. Conclusions

As a concluding remark, we would like to highlight how the technique presented in this paper is only the last step of a complex approach, aimed at inferring artful processes from e-mail messages; once that other techniques, out of the scope of this paper, allow us to consider e-mail messages as strings over an alphabet of characters, the MINERful technique presented in this paper is able to infer which constraints are valid over such strings, thus inferring the process (described in a declarative way) that may lay behind them.

Further research activities are needed in order to refine and solve all the techniques of MailOfMine, and an extensive validation of the overall approach. In this paper, we have shown that MINERful is a very efficient algorithm for mining constraints over strings, but only through a validation over real sets of e-mail messages we can really assess how much underfitting or overfitting is the technique. Indeed, we aim at validating it on a corpus of about 10 Gigabyte of e-mail messages, derived from the activity of one of the authors in about 10 years of works in research projects, in order to infer common processes that partners adopted during software/deliverables' production. Then we will apply to the field of collaborative activities of Open Source software development, reported by publicly available mailing lists.

References

[1] P. Warren, N. Kings, I. Thurlow, J. Davies, T. Buerger, E. Simperl, C. Ruiz, J. M. Gomez-Perez, V. Ermolayev, R. Ghani, M. Tilly, T. Bösser, and A. Imtiaz, "Improving knowledge worker productivity - the Active integrated approach," *BT Technologiy Journal*, vol. 26, no. 2, pp. 165–176, 2009.

[2] T. Catarci, A. Dix, A. Katifori, G. Lepouras, and A. Poggi, "Task-centred information management," in *DELOS Conference*, ser. Lecture Notes in Computer Science, vol. 4877. Springer, 2007, pp. 197–206.

[3] P. Innocenti, S. Ross, E. Maceciuvite, T. Wilson, J. Ludwig, and W. Pempe, "Assessing digital preservation frameworks: the approach of the SHAMAN project," in *MEDES*, R. Chbeir, Y. Badr, E. Kapetanios, and A. J. M. Traina, Eds. ACM, 2009, pp. 412–416.

[4] Smart Vortex Consortium, "Smart Vortex – Management and analysis of massive data streams to support large-scale collaborative engineering projects," FP7 IP Project. [Online]. Available: http://www.smartvortex.eu/

[5] D. Heutelbeck, "Preservation of enterprise engineering processes by social collaboration software," 2011, personal communication.

[6] C. Di Ciccio, M. Mecella, M. Scannapieco, D. Zardetto, and T. Catarci, "MailOfMine – analyzing mail messages for mining artful collaborative processes," in *Proceedings of the 1st International Symposium on Data-Driven Process Discovery and Analysis (SIMPDA 2011)*, K. Aberer, E. Damiani, and T. Dillon, Eds., June-July 2011, pp. 45–59. [Online]. Available: http://www.dis.uniroma1.it/~cdc/pubs/SIMPDA2011.pdf

[7] W. M. P. van der Aalst, "The application of petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[8] 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 (Demos)*, ser. CEUR Workshop Proceedings, A. K. A. de Medeiros and B. Weber, Eds., vol. 489. CEUR-WS.org, 2009.

[9] W. M. P. van der Aalst, "Verification of workflow nets," in *ICATPN*, ser. Lecture Notes in Computer Science, P. Azéma and G. Balbo, Eds., vol. 1248. Springer, 1997, pp. 407–426.

[10] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *Advances in Database Technology EDBT'98*, ser. Lecture Notes in Computer Science, H.-J. Schek, G. Alonso, F. Saltor, and I. Ramos, Eds. Springer Berlin / Heidelberg, 1998, vol. 1377, pp. 467–483, 10.1007/BFb0101003. [Online]. Available: http://dx.doi.org/10.1007/BFb0101003

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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, 10.1007/s10270-008-0106-z. [Online]. Available: http://dx.doi.org/10.1007/s10270-008-0106-z

[16] W. M. P. van der Aalst and M. Pesic, "Decserflow: Towards a truly declarative service flow language," in *WS-FM*, ser. Lecture Notes in Computer Science, M. Bravetti, M. Núñez, and G. Zavattaro, Eds., vol. 4184. Springer, 2006, pp. 1–23.

[17] M. Pesic and W. M. P. van der Aalst, "A declarative approach for flexible business processes management," in *Business Process Management Workshops*, ser. Lecture Notes in Computer Science, J. Eder and S. Dustdar, Eds., vol. 4103. Springer, 2006, pp. 169–180.

[18] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst, "User-guided discovery of declarative process models," in *CIDM*. IEEE, 2011, pp. 192–199.

[19] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst, "Declare: Full support for loosely-structured processes," in *EDOC*. IEEE Computer Society, 2007, pp. 287–300.

[20] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in *OTM Conferences (1)*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 4803. Springer, 2007, pp. 77–94.

[21] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *PSTV*, ser. IFIP Conference Proceedings, P. Dembinski and M. Sredniawa, Eds., vol. 38. Chapman & Hall, 1995, pp. 3–18.

[22] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *ASE*. IEEE Computer Society, 2001, pp. 412–416.

[23] M. Westergaard, "Better algorithms for analyzing and enacting declarative workflow languages using ltl," in *BPM*, ser. Lecture Notes in Computer Science, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., vol. 6896. Springer, 2011, pp. 83–98.

[24] 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.

[25] 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, 2008.

[26] V. R. de Carvalho and W. W. Cohen, "Learning to extract signature and reply lines from email," in *CEAS*, 2004.

[27] J. Searle, *A Taxonomy of Illocutionary Acts*. Minneapolis: University of Minnesota Press, 1975, pp. 334–369.

[28] W. W. Cohen, V. R. Carvalho, and T. M. Mitchell, "Learning to classify email into "speech acts"," in *EMNLP*. ACL, 2004, pp. 309–316.
[29] H. Weigand and A. de Moor, "Workflow analysis with communication norms," *Data Knowl. Eng.*, vol. 47, no. 3, pp. 349–369, 2003.
[30] M. N. Garofalakis, R. Rastogi, and K. Shim, "Spirit: Sequential pattern mining with regular expression constraints," in *VLDB*, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 223–234.