Techniques for A Posteriori Analysis of Declarative Processes

Andrea Burattin
University of Padua
burattin@math.unipd.it

Fabrizio M. Maggi
Eindhoven University of Technology
f.m.maggi@tue.nl

Wil M. P. van der Aalst
Eindhoven University of Technology
w.m.p.v.d.aalst@tue.nl

Alessandro Sperduti
University of Padua
sperduti@math.unipd.it

***Abstract*—The increasing availability of event data recorded by information systems, electronic devices, web services and sensor networks provides detailed information about the *actual* processes in systems and organizations. Process mining techniques can use such event data to discover processes and check the conformance of process models. For conformance checking, we need to analyze whether the observed behavior matches the modeled behavior. In such settings, it is often desirable to specify the expected behavior in terms of a *declarative* process model rather than of a detailed procedural model. However, declarative models do not have an explicit notion of state, thus making it more difficult to pinpoint deviations and to explain and quantify discrepancies. This paper focuses on providing high-quality and understandable diagnostics. The notion of *activation* plays a key role in determining the effect of individual events on a given constraint. Using this notion, we are able to show cause-and-effect relations and measure the healthiness of the process.**

***Keywords*-process mining; conformance checking; Declare; temporal logic;**

I. Introduction

Imperative process modeling languages such as BPMN, Petri nets, UML ADs, EPCs and BPEL, are very useful in environments that are stable and where the decision procedures can be predefined. Participants can be guided based on such process models. However, they are less appropriate for environments that are more variable and that require more flexibility. Consider, for instance, a physician in a hospital confronted with a variety of patients that need to be handled in a flexible manner. Nevertheless, there are some general regulations and guidelines to be followed. In such cases, *declarative* process models are more effective than the imperative ones [1], [2], [3]. Instead of explicitly specifying all possible sequences of activities in a process, declarative models implicitly specify the allowed behavior of the process with constraints, i.e., rules that must be followed during execution. In comparison to imperative approaches, which produce "closed" models (what is not explicitly specified is forbidden), declarative languages are "open" (everything that is not forbidden is allowed). In this way, models offer flexibility and still remain compact.

Declarative languages based on *LTL* (Linear Temporal Logic) [4] can be fruitfully applied in the context of process discovery [5], [6], [7] and compliance checking [8], [9], [10], [11], [12]. In [3], [13], the authors introduce an LTL-based declarative process modeling language called *Declare*. Declare is characterized by a user-friendly graphical representation with formal semantics grounded in LTL. A Declare model is a set of Declare constraints, which are defined as instantiations of Declare templates. Templates are abstract entities that define parameterized classes of properties. The example in Figure 1 shows the representation of the *response* template $\Box(a \Rightarrow \Diamond b)$ in Declare and its possible instantiation, where parameters a and b take the values $Create\ Questionnaire$ and $Send\ Questionnaire$. This constraint means that every action $Create\ Questionnaire$ must eventually be followed by action $Send\ Questionnaire$. The above characteristics make Declare very suitable for defining and analyzing compliance models, i.e., checking whether the behavior of a system (e.g., recorded in an event log of the system) complies with predefined regulations.

While in imperative languages, designers tend to forget incorporating some possible scenarios (e.g., related to exception handling), in declarative languages, designers tend to forget certain constraints. This leads to underspecification rather than overspecification, i.e., people are expected to act responsibly and are free to select scenarios that may seem out-of-the-ordinary at first sight.

When analyzing the conformance of a process with respect to a set of constraints, it is important to note that constraints can be vacuously satisfied. For example, if $Create\ Questionnaire$ never occurs, then the *response* constraint holds trivially. This is commonly referred to as *vacuous satisfaction*. In this paper, we start from the existing notion of vacuity detection [14], [15], [16] and we propose an approach for evaluating the "degree of adherence" of a process trace with respect to a Declare model. In particular, we introduce the notion of *healthiness* of a trace that is, in turn, based on the concept of *activation* of a Declare constraint.

An activation of a constraint in a trace is an event whose occurrence imposes, because of that constraint, some obligations on other events in the same trace.

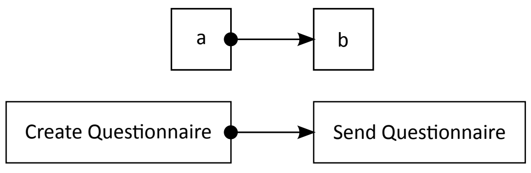


Figure 1: Response template and its possible instantiation

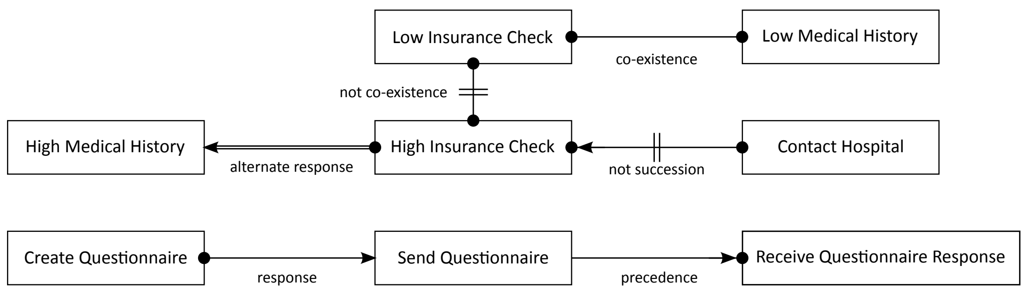


Figure 2: Running example: Declare model consisting of six constraints and eight activities

In the example in Figure 1, the occurrence of $Create$ $Questionnaire$ imposes that $Send$ $Questionnaire$ must eventually occur. An activation is a fulfillment or a violation depending on whether the imposed obligation is fulfilled or not. Consider, for instance, trace $\langle Create\ Questionnaire,\ Send\ Questionnaire,\ Create\ Questionnaire,\ Contact\ Hospital\rangle$. The obligation imposed by the first occurrence of $Create$ $Questionnaire$ is fulfilled ($Send$ $Questionnaire$ eventually occurs after $Create$ $Questionnaire$), whereas the obligation imposed by the second occurrence of $Create$ $Questionnaire$ is not ($Send$ $Questionnaire$ does not occur after the second $Create$ $Questionnaire$). Therefore, we consider the first activation of the *response* constraint in this trace a fulfillment and the second one a violation.

The healthiness of a process trace can be quantified based on the number of activations, fulfillments and violations of the given constraint in the trace. In this paper, we define a set of "health indicators" to measure the healthiness of the trace with respect to that constraint. All conformance checking techniques described in this paper have been implemented in ProM and have been evaluated on a variety of synthetic and real-life logs.

The paper is structured as follows. Section II introduces the Declare language through a running example. Section III gives an overview of the techniques we have developed for a posteriori analysis of declarative processes. In Section IV we demonstrate the scalability of our approach through some experiments and report on a case study based on real-life logs from several Dutch municipalities. Section V concludes the paper.

II. Declare

In this paper, we present an approach for a posteriori analysis of Declare processes based on event logs. Therefore, we first introduce *Declare*[1] [3], [17], [13] through a running example.

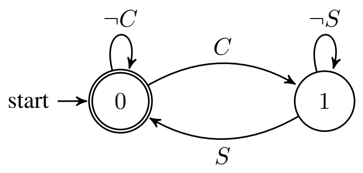
The insurance claim process that we use as our running example corresponds to the handling of health insurance claims in a travel agency. A claim can be classified as high or low but not both. For low claims, two independent tasks, i.e., check insurance and check medical history need to be both executed. Also for high claims, an insurance check is needed. However, in the case of high claims, an insurance check is always followed by a medical history check. It is possible to check the insurance more than once but it is not possible to execute the high insurance check twice without a medical history check in between. It is also possible to contact doctor/hospital for verification. However, in case of high claims, this cannot be done before the insurance check. If one of the checks shows that the claim is not valid, then the claim is rejected; otherwise, it is accepted. In this process, questionnaires can also be created and sent to the applicant. The applicant can decide whether to fill in the questionnaires or not.

Figure 2 shows a simple Declare model with some example constraints for the insurance claim process just described. The model includes eight activities (depicted as rectangles, e.g., *Create Questionnaire*) and six constraints (shown as connectors between the activities, e.g., *not co-existence*). The *not co-existence* constraint indicates that *Low Insurance Check* and *High Insurance Check* can never coexist in the same trace. On the other hand, the *co-existence* constraint indicates that if *Low Insurance Check* and *Low Medical History* occur in a trace, they always coexist. If *High Medical History* is executed, *High Insurance Check* is eventually executed without other occurrences of *High Medical History* in between. This is specified by the *alternate response* constraint. Moreover, the *not succession* constraint means that *Contact Hospital* cannot be followed by *High Insurance Check*. The *precedence* constraint indicates that, if *Receive Questionnaire Response* is executed, *Send Questionnaire* must be executed before (but if *Send Questionnaire* is executed this is not necessarily followed by *Receive Questionnaire Response*). Finally, if *Create Questionnaire* is executed this is eventually followed by *Send Questionnaire* as indicated by the *response* constraint. In the following, we indicate activities *Create Questionnaire*, *Send Questionnaire*, *Receive Questionnaire Response*, *High Insurance Check*, *Low Insurance Check* and *High Medical History* with C, S, R, H, L and M respectively.

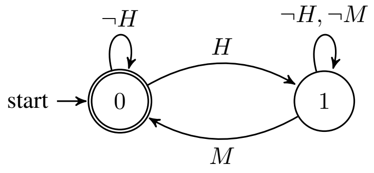
Declare is grounded in *Linear Temporal Logic* (LTL) with a finite-trace semantics[2]. For instance, a constraint like the *response* constraint in Figure 2 can be formally represented using LTL and in particular, it can be written as $\Box(C \Rightarrow \Diamond S)$ that means "whenever activity *Create*

[1]http://www.win.tue.nl/declare/

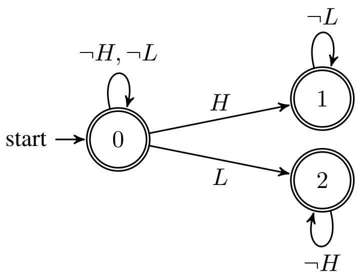
[2]For compactness, in the following we will use the LTL acronym to denote LTL on finite traces.



(a) *response* constraint



(b) *alternate response* constraint



(c) *not co-existence* constraint

Figure 3: Automata for the *response*, *alternate response* and *not co-existence* constraints in our running example

Questionnaire is executed, eventually activity *Send Questionnaire* is executed". In a formula like this, it is possible to find traditional logical operators (e.g., implication $\Rightarrow$), but also temporal operators characteristic of LTL (e.g., always $\Box$ and eventually $\Diamond$). In general, using the LTL language it is possible to express constraints relating activities (atoms) through logical operators or temporal operators.

The logical operators are: implication ($\Rightarrow$), conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). The main temporal operators are: *always* ($\Box p$, in every future state p holds), *eventually* ($\Diamond p$, in some future state p holds), *next* ($\bigcirc p$, in the next state p holds) and *until* ($p \sqcup q$, p holds until q holds).

However, LTL constraints are not very readable for non-experts. Therefore, Declare provides an intuitive graphical front-end for LTL formulas. The LTL back-end of Declare allows us to verify Declare constraints and Declare models, i.e., sets of Declare constraints.

For instance, a Declare constraint can be verified on a log by translating its LTL semantics into a finite state automaton [18] that we call *constraint automaton*. Figure 3 depicts the constraint automata for the *response* constraint, the *alternate response* constraint and the *not co-existence* constraint in Figure 2. In all cases, state 0 is the initial state and accepting states are indicated using a double outline. A transition is labeled with the activity triggering it. As well as positive labels, we also have negative labels (e.g., $\neg L$ for state 0 of the *not co-existence* constraint). This indicates that we can follow the transition for any event not mentioned (e.g., we can execute event *C* from state 0 of the *not co-existence* automaton and remain in the same state). This allows us to use the same automaton regardless of the input language. A constraint automaton *accepts a trace* (i.e., the LTL formula holds) if and only if there exists a corresponding path that starts in the initial state and ends in an accepting state.

III. An Approach for A Posteriori Analysis

In this section, we first define what an "activation", a "fulfillment", a "violation" and a "conflict" are for a Declare constraint. After that, we present an algorithm to identify them in a process trace. Finally, we show how we can define various "health indicators" based on these notions. These indicators are used to evaluate the degree of healthiness of a process trace and of a log.

A. Vacuity Detection in Declare

In [6], the authors introduce for the first time the concept of vacuity detection for Declare constraints. Consider, for instance, the *response* constraint in Figure 2. This constraint is satisfied when a questionnaire is created and is (eventually) sent. However, this constraint is also satisfied in cases where the questionnaire is not created at all. In this latter case, we say that the constraint is *vacuously satisfied*. Cases where a constraint is *non*-vacuously satisfied are called *interesting witnesses* for that constraint.

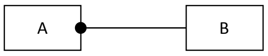
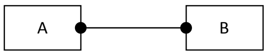
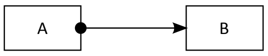
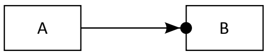
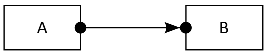
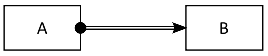
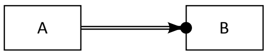
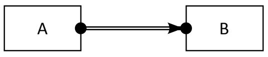
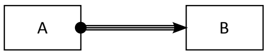
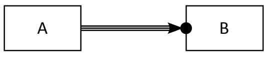
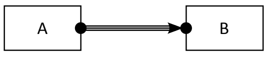
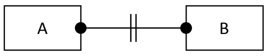
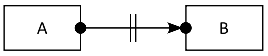
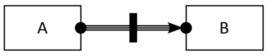
In [15], the authors introduce an approach for vacuity detection in temporal model checking for LTL; they provide a method for extending an LTL formula φ to a new formula $witness(\varphi)$ that, when satisfied, ensures that the original formula φ is non-vacuously satisfied. In particular, $witness(\varphi)$ is generated by considering that a path π satisfies φ non-vacuously (and then is an interesting witness for φ), if π satisfies φ and π satisfies a set of additional conditions that guarantee that every subformula of φ does really affect the truth value of φ in π. We call these conditions *vacuity detection conditions* of φ. They correspond to the formulas $\neg\varphi[\psi \leftarrow \bot]$, where, for all the subformulas ψ of φ, $\varphi[\psi \leftarrow \bot]$ is obtained from φ by replacing ψ by false or true, depending on whether ψ is in the scope of an even or an odd number of negations. Then, $witness(\varphi)$ is the conjunction of φ and all the formulas $\neg\varphi[\psi \leftarrow \bot]$ with ψ subformula of φ:

$$witness(\varphi) = \varphi \wedge \bigwedge \neg\varphi[\psi \leftarrow \bot]. \qquad (1)$$

Consider, for instance, the *response* constraint $\Box(C \Rightarrow \Diamond S)$. In this case, the vacuity detection condition is $\Diamond C$, so that the interesting witnesses for this constraint are all cases where $\Box(C \Rightarrow \Diamond S) \wedge \Diamond C$ is valid. Roughly speaking, the constraint specifying that "whenever activity *Create Questionnaire* is executed, eventually activity *Send Questionnaire* is executed" is non-trivially valid in all cases where it is valid and where, also, *Create Questionnaire* is actually executed.

This approach was applied to Declare in [6] for vacuity detection in the context of process discovery. However,

Table I: Semantics for Declare constraints, activations and graphical representation

Name	Constraint	Activations	Graphical representation
Relation templates			
$responded\,existence(A,B)$	$\Diamond A \Rightarrow \Diamond B$	A	
$co\text{-}existence(A,B)$	$\Diamond A \Leftrightarrow \Diamond B$	A, B	
$response(A,B)$	$\Box(A \Rightarrow \Diamond B)$	A	
$precedence(A,B)$	$(\neg B \sqcup A) \vee \Box(\neg B)$	B	
$succession(A,B)$	$response(A,B) \wedge precedence(A,B)$	A, B	
$alternate\ response(A,B)$	$\Box(A \Rightarrow \bigcirc(\neg A \sqcup B))$	A	
$alternate\,precedence(A,B)$	$precedence(A,B) \wedge \Box(B \Rightarrow \bigcirc(precedence(A,B)))$	B	
$alternate\ succession(A,B)$	$alternate\ response(A,B) \wedge alternate\ precedence(A,B)$	A, B	
$chain\ response(A,B)$	$\Box(A \Rightarrow \bigcirc B)$	A	
$chain\ precedence(A,B)$	$\Box(\bigcirc B \Rightarrow A)$	B	
$chain\ succession(A,B)$	$\Box(A \Leftrightarrow \bigcirc B)$	A, B	
Negative relation templates			
$not\ co\text{-}existence(A,B)$	$\neg(\Diamond A \wedge \Diamond B)$	A, B	
$not\ succession(A,B)$	$\Box(A \Rightarrow \neg(\Diamond B))$	A, B	
$not\,chain\ succession(A,B)$	$\Box(A \Rightarrow \bigcirc(\neg B))$	A, B	

the algorithm introduced in [15] can generate different results for equivalent LTL formulas. Consider, for instance, the following equivalent formulas (corresponding to the *alternate response* constraint in our running example):

$$\varphi = \Box(H \Rightarrow \Diamond M) \wedge \Box(H \Rightarrow \bigcirc((\neg H \sqcup M) \vee \Box(\neg M))),$$
$$\varphi' = \Box(H \Rightarrow \bigcirc(\neg H \sqcup M)).$$

When we apply (1) to φ and φ', we obtain that $witness(\varphi) \neq witness(\varphi')$:

$$witness(\varphi) = \mathit{false},$$
$$witness(\varphi') = \varphi' \wedge \Diamond(\neg \bigcirc (\neg H \sqcup M)) \wedge \Diamond(H) \wedge \Diamond(H \wedge \neg \bigcirc (M)).$$

In compliance models, LTL-based declarative languages like Declare are used to describe requirements to the process behavior. In this case, each LTL rule describes a specific constraint with clear semantics. Therefore, we need a *univocal* (i.e., not sensitive to syntax) and intuitive way to diagnose vacuously compliant behavior in an LTL-based process model. Furthermore, interesting witnesses for a Declare constraint could show very different behaviors. Consider, for instance, the *response* constraint $\Box(C \Rightarrow \Diamond S)$ and traces $p_1 = \langle C,S,C,S,C,S,C,S,R\rangle$ and $p_2 = \langle H,M,C,S,H,M,R\rangle$. Both p_1 and p_2 are interesting witnesses for $\Box(C \Rightarrow \Diamond S)$ (in both traces $\Box(C \Rightarrow \Diamond S)$ is valid and the vacuity detection condition $\Diamond C$ is also valid). However, it is intuitive to understand that in p_1 this constraint is activated four times (because *Create Questionnaire* occurs four times), whereas in p_2 it is activated only once. To solve these issues we introduce the notion of *constraint activation*.

Definition 1 (Subtrace): Let σ be a trace. A trace σ' is a *subtrace* of σ ($\sigma' \sqsubset \sigma$) if σ' can be obtained from σ by removing one or more events.

Definition 2 (Minimal Violating Trace): Let π be a Declare constraint and $\mathcal{A}_\pi$ the constraint automaton of π. A trace σ is a *minimal violating trace* for $\mathcal{A}_\pi$ if it is not accepted by $\mathcal{A}_\pi$ and if every subtrace of σ is accepted by $\mathcal{A}_\pi$.

Definition 3 (Constraint Activation): Let π be a Declare constraint and $\mathcal{A}_\pi$ the constraint automaton of π. Each *event* included in a minimal violating trace for $\mathcal{A}_\pi$ is an *activation* of π.

Consider, for instance, the automaton in Figure 3(a). In this case, the minimal violating trace is $\langle C\rangle$. Therefore, the *response* constraint in our running example is activated by *Create Questionnaire*. Moreover, for the automaton in Figure 3(b), the minimal violating trace is $\langle H\rangle$ and, then, the *alternate response* constraint is activated by *High Insurance Check*. Finally, for the automaton in Figure 3(c), the minimal violating sequences are $\langle L,H\rangle$ and $\langle H,L\rangle$.

The *not co-existence* constraint is, therefore, activated by both *High Insurance Check* and *Low Insurance Check*.

Roughly speaking, an activation for a constraint is an event that constrains in some way the behavior of other events and imposes some obligations on them. For instance, the occurrence of an event can require the occurrence of another event afterwards (e.g., in the *response* constraint) or beforehand (e.g., in the *precedence* constraint). An obligation can also be a prohibition. Consider, for instance, the *not co-existence* constraint in Figure 2. In this case, the occurrence of one of the two events *High Insurance Check* and *Low Insurance Check* forbids the occurrence of the other one.

In Table I, we indicate events that represent an activation for each Declare constraint. Note that events that represent an activation for a constraint are marked with a black dot in the graphical notation of Declare, e.g., both A and B are activations for the *succession* constraint (as visualized by the black dots).

B. An Algorithm to Discriminate Fulfillments from Violations

When a trace is compliant w.r.t. a constraint, every activation of that constraint leads to a fulfillment. For instance, in p_1, the *response* constraint is activated and fulfilled four times, whereas in p_2, the same constraint is activated and fulfilled once. Notice that, when a trace is non-compliant w.r.t. a constraint, an activation of a constraint can lead to a fulfillment but also to a violation (and at least one activation leads to a violation). Consider, again, the *response* constraint in our running example and the trace $p_3 = \langle C, S, C, R\rangle$. In this trace, the *response* constraint is violated. However, it is still possible to quantify the degree of adherence of this trace in terms of number of fulfillments and violations. Indeed, in this case, the *response* constraint is activated twice, but one activation leads to a fulfillment (eventually an event S occurs) and one activation leads to a violation (S does not occur eventually). Therefore, we need a mechanism to point out that the first occurrence of $Create\ Questionnaire$ is a fulfillment and the second one is a violation.

Furthermore, if we consider trace $\langle H, H, M\rangle$ and the *alternate response* constraint in our running example, we have that the two occurrences of H cannot co-exist but it is impossible to understand (without further information from the user) which one is a violation and which one is a fulfillment. In this case, we say that we have a *conflict* between the two activations.

In order to identify fulfillments, violations and conflicts for a constraint π in a trace σ, we present now an algorithm (see Algorithm 1) that is based on the construction of a so-called *activation tree* of σ w.r.t. π, where every node is labeled with a subtrace of σ. The algorithm starts from a root labeled with the empty subtrace. Then, σ is replayed and the tree is built in the following way:

- if the current event in σ is an activation of π, two children are appended to each leaf-node: a left child labeled with the subtrace of the parent node and a right child labeled with the same subtrace augmented with the current activation;
- if the current event in σ is not an activation of π, all leaf-nodes are augmented with the current event.

Algorithm 1: Procedure to build the activation tree

```
Input: $\sigma$: trace; $\pi$: constraint
Result: activation tree of $\sigma$ w.r.t. $\pi$
1  Let $T$ be a binary tree with root labeled with an empty subtrace
2  forall the $e \in \sigma$ (explored in sequence) do
3      forall the leaf $l$ of $T$ do
4          if the subtrace associated to $l$ is not dead then
5              if $e$ is an activation for $\pi$ then
6                  $l[left]$ = new node, subtrace of $l$
7                  $l[right]$ = new node, subtrace of $l$ + $e$
8              else
9                  subtrace of $l$ = subtrace of $l$ + $e$
10             end
11         end
12     end
13 end
14 return $T$
```

At each iteration, each subtrace in the leaf-nodes is executed on the constraint automaton $\mathcal{A}_\pi$. A node is called *dead* if the corresponding subtrace is not possible according to the automaton or all events have been explored and no accepting state has been reached. Dead nodes are not explored further and crossed-out in the diagrams.

At the end of the algorithm, fulfillments, violations and conflicts can be identified by selecting, among the (non-dead) leaf-nodes, the *maximal fulfilling subtraces*.

Definition 4 (Maximal Subtrace): Given a set Σ of subtraces of a trace σ, a *maximal subtrace* of σ in Σ is an element $\sigma' \in \Sigma$ such that $\nexists \sigma'' \in \Sigma$ with $\sigma' \sqsubset \sigma''$.

Definition 5 (Maximal Fulfilling Subtrace): Given a trace σ and a constraint π, let $\overline{\Sigma}$ be the set of the

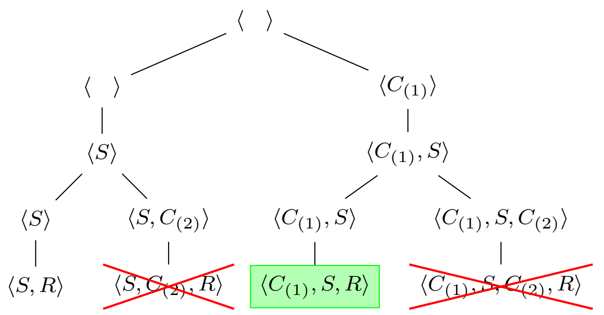


Figure 4: Activation tree of trace $\langle C_{(1)}, S, C_{(2)}, R\rangle$ w.r.t. the *response* constraint in our running example: dead nodes are crossed out and nodes corresponding to maximal fulfilling subtraces are highlighted

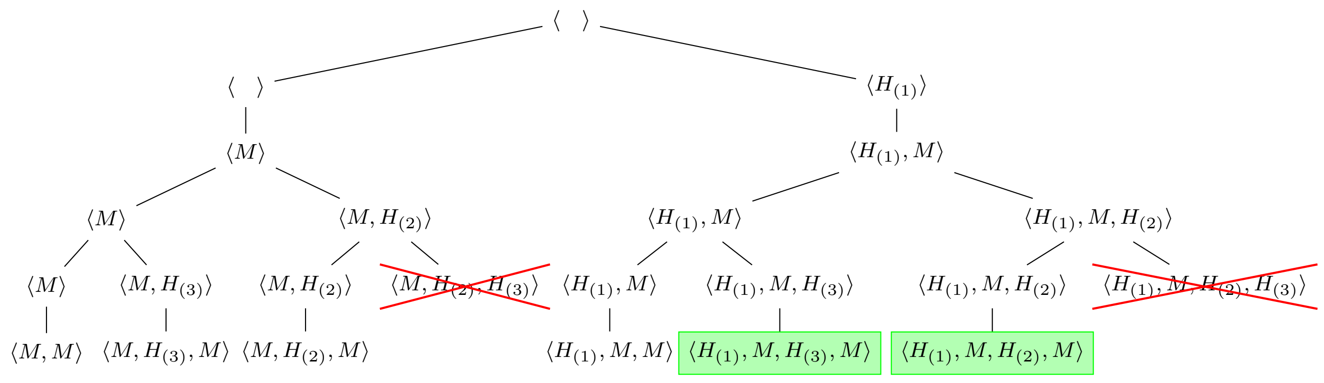

$\langle\ \rangle$
$\langle\ \rangle$
$\langle H_{(1)} \rangle$
$\langle M \rangle$
$\langle H_{(1)}, M \rangle$
$\langle M \rangle$
$\langle M, H_{(2)} \rangle$
$\langle H_{(1)}, M \rangle$
$\langle H_{(1)}, M, H_{(2)} \rangle$
$\langle M \rangle$
$\langle M, H_{(3)} \rangle$
$\langle M, H_{(2)} \rangle$
$\langle M, H_{(2)}, H_{(3)} \rangle$
$\langle H_{(1)}, M \rangle$
$\langle H_{(1)}, M, H_{(3)} \rangle$
$\langle H_{(1)}, M, H_{(2)} \rangle$
$\langle H_{(1)}, M, H_{(2)}, H_{(3)} \rangle$
$\langle M, M \rangle$
$\langle M, H_{(3)}, M \rangle$
$\langle M, H_{(2)}, M \rangle$
$\langle H_{(1)}, M, M \rangle$
$\langle H_{(1)}, M, H_{(3)}, M \rangle$
$\langle H_{(1)}, M, H_{(2)}, M \rangle$

Figure 5: Activation tree of trace $\langle H_{(1)}, M, H_{(2)}, H_{(3)}, M \rangle$ w.r.t. the *alternate response* constraint in our running example

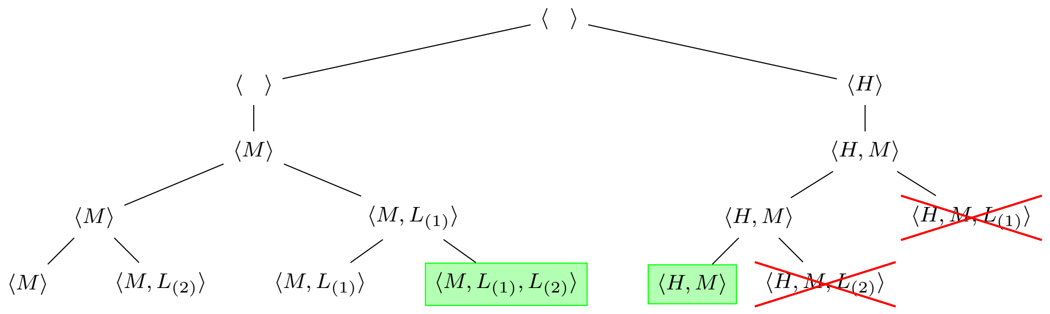

$\langle\ \rangle$
$\langle\ \rangle$
$\langle H \rangle$
$\langle M \rangle$
$\langle H, M \rangle$
$\langle M \rangle$
$\langle M, L_{(1)} \rangle$
$\langle H, M \rangle$
$\langle H, M, L_{(1)} \rangle$
$\langle M \rangle$
$\langle M, L_{(2)} \rangle$
$\langle M, L_{(1)} \rangle$
$\langle M, L_{(1)}, L_{(2)} \rangle$
$\langle H, M \rangle$
$\langle H, M, L_{(2)} \rangle$

Figure 6: Activation tree of trace $\langle H, M, L_{(1)}, L_{(2)} \rangle$ w.r.t. the *not co-existence* constraint in our example

subtraces of σ associated to the non-dead leaf-nodes of the activation tree of σ w.r.t. π. Let $M \subseteq \overline{\Sigma}$ the set of the maximal subtraces of σ in $\overline{\Sigma}$. An element of M is called maximal fulfilling subtrace of σ w.r.t. π.

An activation a of π in σ is a fulfillment if a is included in all the maximal fulfilling subtraces of σ w.r.t. π; a is a violation if a is not included in any maximal fulfilling subtrace of σ w.r.t. π; a is a conflict if a is only included in some maximal fulfilling subtraces of σ w.r.t. π.

Consider, for instance, the activation tree in Figure 4 of trace $\langle C_{(1)}, S, C_{(2)}, R \rangle$ w.r.t. the *response* constraint in our running example. The maximal fulfilling subtrace is $\langle C_{(1)}, S, R \rangle$. We can conclude that $C_{(1)}$ is a fulfillment, whereas $C_{(2)}$ is a violation. Figure 5 depicts the activation tree of trace $\langle H_{(1)}, M, H_{(2)}, H_{(3)}, M \rangle$ w.r.t. the *alternate response* constraint in our running example. The maximal fulfilling subtraces are, in this case, $\langle H_{(1)}, M, H_{(2)}, M \rangle$ and $\langle H_{(1)}, M, H_{(3)}, M \rangle$. We can conclude that $H_{(1)}$ is a fulfillment, whereas $H_{(2)}$ and $H_{(3)}$ are conflicts. Finally, Figure 6 depicts the activation tree of trace $\langle H, M, L_{(1)}, L_{(2)} \rangle$ w.r.t. the *not co-existence* constraint in our running example. The maximal fulfilling subtraces are, in this case, $\langle H, M \rangle$ and $\langle M, L_{(1)}, L_{(2)} \rangle$. We can conclude that H, $L_{(1)}$ and $L_{(2)}$ are conflicts.

C. Healthiness

We now give a definition of the healthiness of a trace with respect to a given constraint. Given a trace σ and constraint π, each event in the trace can be classified as an activation or not based on Def. 3. $n_a(\sigma, \pi)$ is the number of activations of σ w.r.t. π. Each activation can be classified as a fulfillment, a violation, or a conflict based on the activation tree. $n_f(\sigma, \pi)$, $n_v(\sigma, \pi)$ and $n_c(\sigma, \pi)$ denote the numbers of fulfillments, violations and conflicts of σ w.r.t. π, respectively. $n(\sigma)$ is the number of events in σ.

Definition 6 (Healthiness): The *healthiness* of a trace σ w.r.t. a constraint π is a quadruple $\mathcal{H}_\pi(\sigma) = (AS_\pi(\sigma), FR_\pi(\sigma), VR_\pi(\sigma), CR_\pi(\sigma))$, where:

1) $AS_\pi(\sigma) = 1 - \frac{n_a(\sigma,\pi)}{n(\sigma)}$ is the *activation sparsity* of σ w.r.t. π,
2) $FR_\pi(\sigma) = \frac{n_f(\sigma,\pi)}{n_a(\sigma,\pi)}$ is the *fulfillment ratio* of σ w.r.t. π,
3) $VR_\pi(\sigma) = \frac{n_v(\sigma,\pi)}{n_a(\sigma,\pi)}$ is the *violation ratio* of σ w.r.t. π and
4) $CR_\pi(\sigma) = \frac{n_c(\sigma,\pi)}{n_a(\sigma,\pi)}$ is the *conflict ratio* of σ w.r.t. π.

A trace σ is "healthy" with respect to a constraint π if the fulfillment ratio $FR_\pi(\sigma)$ is high and the violation ratio $VR_\pi(\sigma)$ and the conflict ratio $CR_\pi(\sigma)$ are low. If $FR_\pi(\sigma)$ is high, the activation sparsity $AS_\pi(\sigma)$ becomes a positive factor, otherwise it is symptom of unhealthiness.

It is possible to average the values of the healthiness over the traces in a log and over the constraints in a Declare model thus obtaining aggregated views of the healthiness of a trace w.r.t. a Declare model, of a log w.r.t. a constraint and of a log w.r.t. a Declare model.

D. Likelihood of a Conflict Resolution

Consider trace $\langle H, M, L_{(1)}, L_{(2)} \rangle$ w.r.t. the *not co-existence* constraint in our running example. The maximal fulfilling subtraces are, in this case, $\langle H, M \rangle$ and

$\langle M, L_{(1)}, L_{(2)} \rangle$ and H, $L_{(1)}$ and $L_{(2)}$ are conflicts. However, the maximal fulfilling subtraces also contain further information. In fact, H is included in one of the maximal fulfilling subtraces and $L_{(1)}$ and $L_{(2)}$ in the other one. This means that $L_{(1)}$ and $L_{(2)}$ can co-exist but both cannot co-exist with H. In this way, we can conclude that either H is a violation and $L_{(1)}$ and $L_{(2)}$ are fulfillments or, vice versa, H is a fulfillment and $L_{(1)}$ and $L_{(2)}$ are violations. We call the corresponding maximal fulfilling subtraces *conflict resolutions*.

The user can decide how to solve a conflict by selecting a conflict resolution. However, it is possible to provide the user with two health indicators that can support her in this decision: the *local likelihood* of a conflict resolution and the *global likelihood* of a conflict resolution.

Definition 7 (Local Likelihood): Let σ' be a conflict resolution of a trace σ w.r.t. a constraint π. Let $n_a(\sigma', \pi)$ and $n_f(\sigma', \pi)$ be the number of activations and fulfillments of a conflict resolution σ', respectively. The *local likelihood* of σ' is defined as $LL(\sigma') = \frac{n_f(\sigma', \pi)}{n_a(\sigma', \pi)}$.

Note that the local likelihood of a conflict resolution is a number in the open interval $(0, 1)$. If we consider again the example described before, we have that $LL(\langle H, M \rangle) = 1/3$ and $LL(\langle M, L_{(1)}, L_{(2)} \rangle) = 2/3$. This means that, more likely, H is a violation and $L_{(1)}$ and $L_{(2)}$ are fulfillments.

In the following definition a Declare model is a pair $\mathcal{D} = (A, \Pi)$, where A is a set of activities and Π is a set of Declare constraints defined over activities in A.

Definition 8 (Global Likelihood): Let $\mathcal{D} = (A, \Pi)$ be a Declare model. Let σ' be a conflict resolution of a trace σ w.r.t. a constraint $\pi \in \Pi$. Let K be the set of the conflicting activations in σ. For each conflicting activation $a \in K$, let $\gamma(a)$ be the percentage of constraints in Π where a is a fulfillment, if a is resolved as a fulfillment in σ', or where a is a violation, if a is resolved as a violation in σ'. The *global likelihood* of σ' is defined as $GL(\sigma') = \frac{\sum_a \gamma(a)}{|K|}$.

The global likelihood of a conflict resolution is a number between 0 and 1. If we consider again the example described before, we have that $GL(\langle H, M \rangle) = 1/6$ and $LL(\langle M, L_{(1)}, L_{(2)} \rangle) = 0$. This means that, from the global point of view, more likely, H is a fulfillment and $L_{(1)}$ and $L_{(2)}$ are violations.

IV. Experiments

For the a posteriori analysis of a log w.r.t. a Declare model, we have implemented the *Declare Analyzer*, a plug-in of the process mining tool ProM[3]. The plug-in takes as input a Declare model and a log and – using the algorithm described in Section III – it provides detailed diagnostics and quantifies the health of each trace (and of the whole log).

In Section IV-A, we evaluate the performance of our approach using both synthetic and real-life logs. Then, in Section IV-B, we validate our approach on a real case study in the context of the CoSeLoG project[4] involving 10 Dutch municipalities. Here, we also present our plug-in and illustrate how the detailed diagnostics are visualized graphically.

A. Scalability

In order to experimentally demonstrate the scalability of our approach, we have performed two experiments[5]. In the first experiment, we verify the scalability of the technique when varying the log size. For this experiment, we have generated a set of synthetic logs by modeling the process described as running example in CPN Tools[6] and by simulating the model. In particular, we used randomly generated logs including $250, 500, 750$ and 1000 traces. The results are presented in Figure 7. The plot shows that

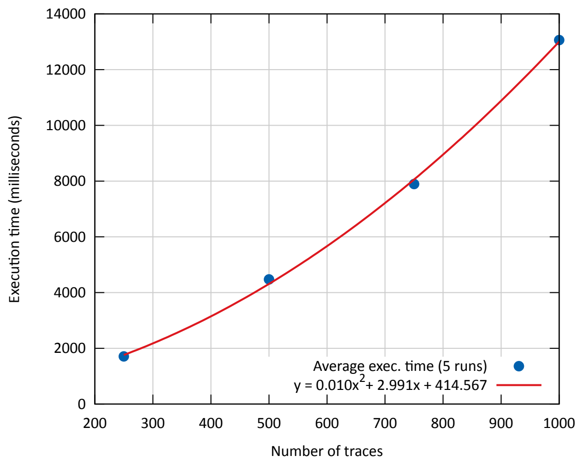


Figure 7: Execution time for varying log sizes and the polynomial regression curve associated

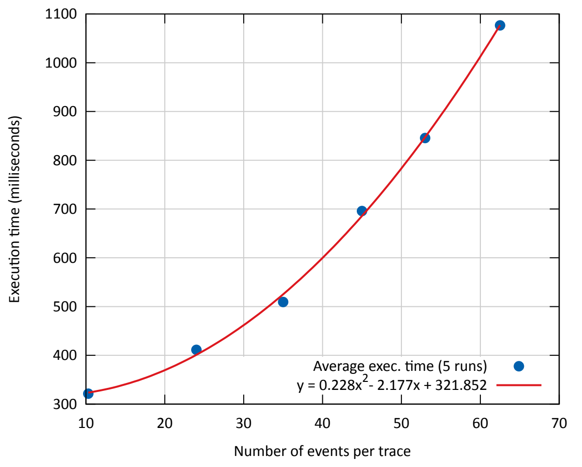


Figure 8: Execution times for varying trace lengths and the polynomial regression curve associated

[3]http://www.promtools.org

[4]http://www.win.tue.nl/coselog

[5]The experiments have been performed on a standard laptop, with a dual-core processor with its power forced to 1.6 GHz. The presented results report the average value of the execution time over 5 runs.

[6]http://cpntools.org

the execution time grows polynomially with the size of the logs.

In the second experiment, we evaluate the trend of the execution time w.r.t. the length of the traces. For this experiment, we have selected, in the CoSeLoG logs, 6 sets of traces, each composed of 10 traces of the same length. Figure 8 shows the results of this experiment. Even if the size of an activation tree is exponential in the number of activations, the execution time is polynomial in the length of the traces. Indeed, performances get worse when the number of activations is close to the number of events in a trace. However, from our experience, in practice, this case is, in general, unlikely. As shown in Section IV-B, the activation sparsity is in most cases high and, therefore, the number of activations is low with respect to the number of events in a trace. This means that, from the practical point of view, the algorithm is applicable. For example, as shown in Figure 8, processing 10 traces with 63 events requires slightly more than 1 second.

In addition, in our implementation we never construct the whole activation tree of a trace. This also influence the performances of the approach. At each step of the algorithm, we keep track only of the maximal traces without building the nodes corresponding to their sub-traces. These sub-traces are identified (and evaluated) only when the original maximal trace is violated (and pruned away).

B. Case study

We have validated our approach by performing various experiments using real-life event logs from the CoSeLoG project. Here, we show results for the process of handling permissions for building or renovating private houses for which we have logs from several Dutch municipalities. For the validation reported here, we have used two logs of processes enacted by two different municipalities. We first have discovered a Declare model using an event log of one municipality using the *Declare Miner* plug-in in ProM [6], [5]. This model is shown in Figure 9. Then, using the *Declare Analyzer*, we have analyzed the degree of adherence of a log of the second municipality with respect to the mined model. Analysis showed commonalities and interesting differences. From a performance viewpoint the results were also encouraging: 481 cases with 17032 events could be replayed in 15 seconds.

Figures 10, 11 and 12 illustrate the output produced by the *Declare Analyzer*. The results are presented both as aggregated details for the entire log (Figure 10) and at the trace level (Figures 11 and 12). From the log view, we can see that the log and the model are reasonably close to each other: the fulfillment ratio is, for almost all constraints, very high and, therefore, the average fulfillment ratio over the entire Declare model is also high (0.7542). The activation sparsity of the log is, in most cases, close to 1, indicating a low activation frequency for each constraint in the model. For the *not succession* constraint between *Beslissing* and *Rapportage*, the combination of an under average activation sparsity with a high fulfillment ratio

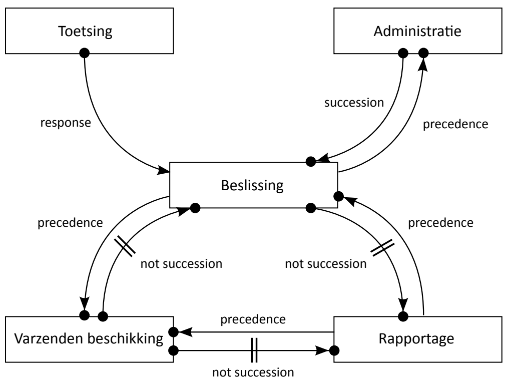


Figure 9: Model discovered from an event log of a Dutch Municipality in the context of the CoSeLoG project. For clarifying, we provide the English translation of the Dutch activity names. *Administratie, Toetsing, Beslissing, Verzenden beschikking* and *Rapportage* can be translated with *Administration, Verification, Judgement, Sending Outcomes* and *Reporting*, respectively

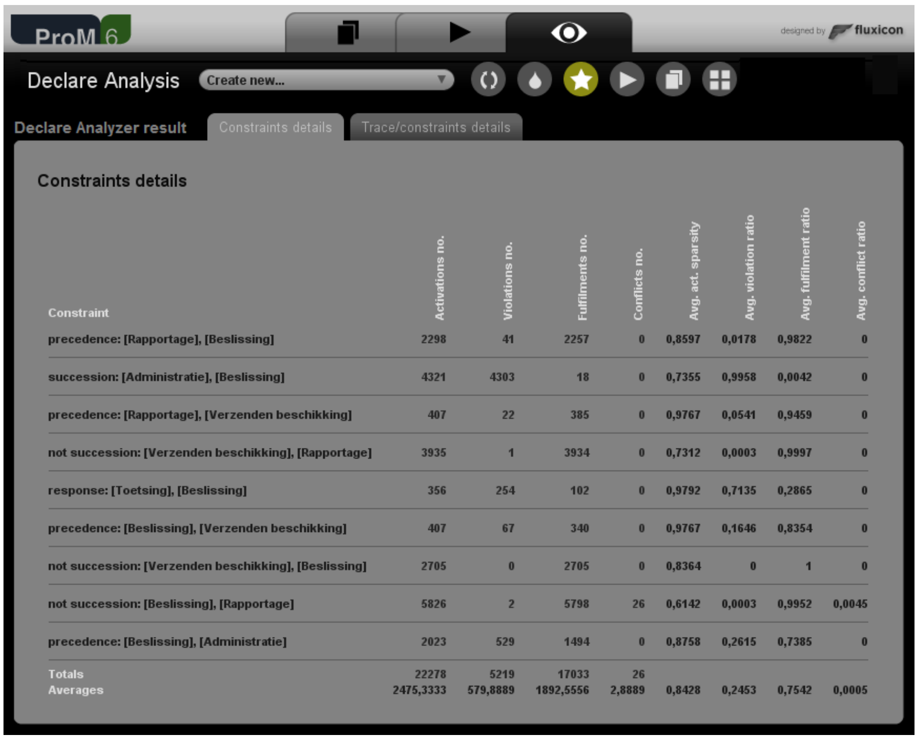
reveals the "good healthiness" of the log with respect to that constraint.

Nevertheless, the two municipalities execute the two processes in a slightly different manner (the violation ratio and the conflict ratio of the log w.r.t. the Declare model are 0.2453 and 0.0005 respectively). The discrepancies have mainly been detected for the *succession* constraint and for the *response* constraint in the reference model. Here, the violation ratio is high. For the *succession* constraint the high violation ratio in combination with a low activation sparsity is symptom of strong unhealthiness.

In the trace representation (Figures 11 and 12), for each trace, aggregated details (averages w.r.t. the entire Declare model) are shown. For trace 22991, we have an activation sparsity of 0.8704, a violation ratio of 0.2143, a fulfillment ratio of 0.6429 and a conflict ratio of 0.1429 w.r.t. the entire model. Different colors are used to indicate violations (red), fulfillments (green) and conflicts (yellow) of each constraint in the trace (events that are not activations are grayed out). For conflicts, the possible resolutions are shown. For every resolution, the local and the global likelihood is indicated (together with their average).

For instance, for trace 22991 in the figure, we have a conflict in the *not succession* constraint between *Beslissing* and *Rapportage* (in this trace, *Rapportage* follows *Beslissing*). The first resolution (with *Beslissing* as a violation and *Rapportage* as a fulfillment) has value 0.5 for the local likelihood and value 0.1111 for the global likelihood (0.3056 on average). The second resolution (with *Beslissing* as a fulfillment and *Rapportage* as a violation) has value 0.5 for the local likelihood and value 0.2222 for the global likelihood (0.3611 on average). Therefore, according to these indicators, it is more likely that *Beslissing* is a fulfillment and *Rapportage* is a violation.

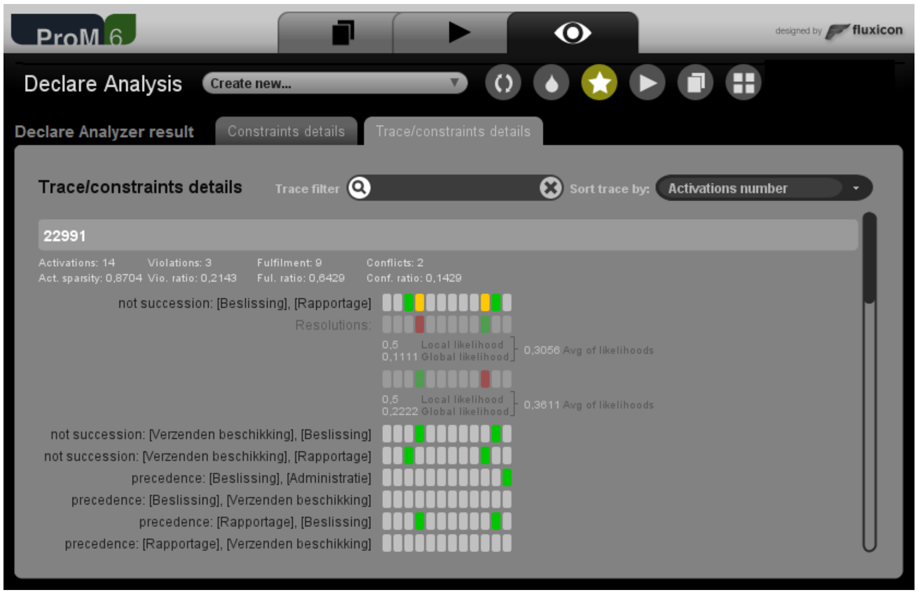
Figure 12 shows that moving with the mouse over a particular constraint it is possible to have details about the healthiness of a trace w.r.t. that specific constraint.



Constraints details

Constraint	Activations no.	Violations no.	Fulfilments no.	Conflicts no.	Avg. act. sparsity	Avg. violation ratio	Avg. fulfilment ratio	Avg. conflict ratio
precedence: [Rapportage], [Beslissing]	2298	41	2257	0	0,8597	0,0178	0,9822	0
succession: [Administratie], [Beslissing]	4321	4303	18	0	0,7355	0,9958	0,0042	0
precedence: [Rapportage], [Verzenden beschikking]	407	22	385	0	0,9767	0,0541	0,9459	0
not succession: [Verzenden beschikking], [Rapportage]	3935	1	3934	0	0,7312	0,0003	0,9997	0
response: [Toetsing], [Beslissing]	356	254	102	0	0,9792	0,7135	0,2865	0
precedence: [Beslissing], [Verzenden beschikking]	407	67	340	0	0,9767	0,1646	0,8354	0
not succession: [Verzenden beschikking], [Beslissing]	2705	0	2705	0	0,8364	0	1	0
not succession: [Beslissing], [Rapportage]	5826	2	5798	26	0,6142	0,0003	0,9952	0,0045
precedence: [Beslissing], [Administratie]	2023	529	1494	0	0,8758	0,2615	0,7385	0
Totals	22278	5219	17033	26				
Averages	2475,3333	579,8889	1892,5556	2,8889	0,8428	0,2453	0,7542	0,0005

Figure 10: Output of the *Declare Analyzer* plug-in: log view.



22991

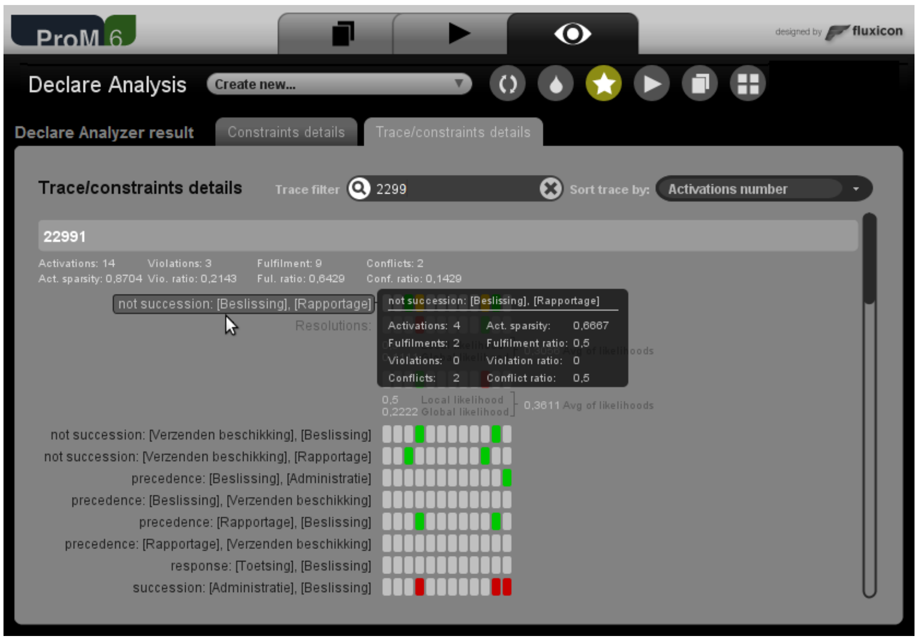
```
Activations: 14          Violations: 3          Fulfilment: 9          Conflicts: 2
Act. sparsity: 0,8704    Vio. ratio: 0,2143     Ful. ratio: 0,6429     Conf. ratio: 0,1429

      not succession: [Beslissing], [Rapportage]
                                    Resolutions:
                                                 0,5    Local likelihood  } 0,3056 Avg of likelihoods
                                                 0,1111 Global likelihood }

                                                 0,5    Local likelihood  } 0,3611 Avg of likelihoods
                                                 0,2222 Global likelihood }
not succession: [Verzenden beschikking], [Beslissing]
not succession: [Verzenden beschikking], [Rapportage]
           precedence: [Beslissing], [Administratie]
   precedence: [Beslissing], [Verzenden beschikking]
              precedence: [Rapportage], [Beslissing]
   precedence: [Rapportage], [Verzenden beschikking]
```

Figure 11: Output of the *Declare Analyzer* plug-in: aggregated trace view.

The activation sparsity of trace 22991 w.r.t. *not succession* constraint between *Beslissing* and *Rapportage* is 0.6667, the violation ratio is 0, while the fulfillment ratio and the conflict ratio are both 0.5.

V. Conclusion and Future Work

This paper presents a novel approach to check the conformance of observed behavior (i.e., an event log) with respect to desired behavior modeled in a declarative manner (i.e., a Declare model). Unlike earlier approaches, we are able to provide reliable diagnostics which do not depend on the underlying LTL syntax. We provided *behavioral* characterizations of *activations*, *fulfillments*, *violations* and *conflicts*. These can be used to provide *detailed diagnostics* at the event level, but can also be aggregated into *health indicators* such as the fulfillment ratio (fraction of activations having no problems), violation ratio and conflict ratio. Experiments show that the approach scales well (polynomial in the size of the log and in the length of the traces). Initial experiences in a case study based on the event logs of two municipalities revealed that the diagnostics are indeed very useful and



```
Trace/constraints details   Trace filter 2299   Sort trace by: Activations number

22991
Activations: 14           Violations: 3          Fulfilment: 9          Conflicts: 2
Act. sparsity: 0,8704     Vio. ratio: 0,2143     Ful. ratio: 0,6429     Conf. ratio: 0,1429

not succession: [Beslissing], [Rapportage]
  Activations:  4    Act. sparsity:     0,6667
  Fulfilments:  2    Fulfilment ratio:  0,5
  Violations:   0    Violation ratio:   0
  Conflicts:    2    Conflict ratio:    0,5

0,5     Local likelihood
0,2222  Global likelihood    0,3611 Avg of likelihoods

not succession: [Verzenden beschikking], [Beslissing]
not succession: [Verzenden beschikking], [Rapportage]
precedence: [Beslissing], [Administratie]
precedence: [Beslissing], [Verzenden beschikking]
precedence: [Rapportage], [Beslissing]
precedence: [Rapportage], [Verzenden beschikking]
response: [Toetsing], [Beslissing]
succession: [Administratie], [Beslissing]
```

Figure 12: Output of the *Declare Analyzer* plug-in: trace view details.

can be interpreted easily.

Acknowledgment

This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

References

[1] S. Zugal, J. Pinggera, and B. Weber, "The impact of test-cases on the maintainability of declarative process models," in *BMMDS/EMMSAD*, 2011, pp. 163–177.

[2] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," in *BPM Workshops*, 2011, pp. 383–394.

[3] W. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative Workflows: Balancing Between Flexibility and Support," *Computer Science - R&D*, pp. 99–113, 2009.

[4] A. Pnueli, "The temporal logic of programs," *FOCS*, pp. 46–57, 1977.

[5] F. M. Maggi, J. Bose, and W. M. P. van der Aalst, "Efficient Discovery of Understandable Declarative Models from Event Logs," in *CAiSE 2012*, 2012, to appear.

[6] F. M. Maggi, A. J. Mooij, and W. M. P. van der Aalst, "User-guided discovery of declarative process models," in *CIDM 2011*, vol. 2725, 2011, pp. 192–199.

[7] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, "Exploiting Inductive Logic Programming Techniques for Declarative Process Mining," *ToPNoC*, vol. 5460, pp. 278–295, 2009.

[8] E. Damaggio, A. Deutsch, R. Hull, and V. Vianu, "Automatic verification of data-centric business processes," in *BPM*, 2011, pp. 3–16.

[9] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[10] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, and P. Dadam, "On enabling data-aware compliance checking of business process models," in *ER 2010*, pp. 332–346.

[11] F. M. Maggi, M. Montali, M. Westergaard, and W. M. P. van der Aalst, "Monitoring business constraints with linear temporal logic: An approach based on colored automata," in *BPM 2011*, vol. 6896, 2011, pp. 132–147.

[12] F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst, "Runtime verification of LTL-based declarative process models," in *RV 2011*, vol. 7186, pp. 131–146.

[13] M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *EDOC 2007*, pp. 287–298.

[14] I. Beer and C. Eisner, "Efficient detection of vacuity in temporal model checking," in *Formal Methods in System Design*, 2001, pp. 200–1.

[15] O. Kupferman and M. Y. Vardi, "Vacuity Detection in Temporal Model Checking," *International Journal on Software Tools for Technology Transfer*, pp. 224–233, 2003.

[16] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning from vacuously satisfiable scenario-based specifications," in *FASE*, 2012, pp. 377–393.

[17] M. Westergaard and F. M. Maggi, "Declare: A tool suite for declarative workflow modeling and enactment," in *BPM (Demos)*, 2011.

[18] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *ASE 2001*, 2001.