Patterns for a Log-Based Strengthening of Declarative Compliance Models

Dennis M.M. Schunselaar[*], Fabrizio M. Maggi[**], and Natalia Sidorova

Eindhoven University of Technology, The Netherlands
{d.m.m.schunselaar,f.m.maggi,n.sidorova}@tue.nl

Abstract. LTL-based declarative process models are very effective when modelling loosely structured processes or working in environments with a lot of variability. A process model is represented by a set of constraints that must be satisfied during the process execution. An important application of such models is compliance checking: a process model defines then the boundaries in which a system/organisation may work, and the actual behaviour of the system, recorded in an event log, can be checked on its compliance to the given model.

A compliance model is often a general one, e.g., applicable for a whole branch of industry, and some constraints used there may be irrelevant for a company in question: for example, a constraint related to property assessment regulations will be irrelevant for a rental agency that does not execute property assessment at all. In this paper, we take the compliance model and the information about past executions of the process instances registered in an event log and, by using a set of patterns, we check which constraints of the compliance model are irrelevant (vacuously satisfied) with respect to the event log. Our compliance patterns are inspired by vacuity detection techniques working on a single trace. However, here we take all the knowledge available in the log into consideration.

Keywords: Linear Temporal Logic, Declare, Vacuity detection, Compliance checking, Event log.

1 Introduction

While imperative process modelling languages such as BPMN, UML ADs, EPCs and BPEL are very useful when it is necessary to provide strong support to the process participants during the process execution, they are less appropriate for environments characterised by high flexibility and variability. In such cases, declarative process models are more effective than the imperative ones [1,14]. Instead of explicitly specifying all the possible sequences of activities in a process,

[*] This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

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

declarative models implicitly specify the allowed behaviour 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.

Recent works have showed that declarative languages based on LTL (Linear Temporal Logic) [16] can be fruitfully applied in the context of process discovery [7,11] and compliance checking [5,10,12]. In [13], the authors introduced a declarative process modelling language called *Declare* characterised by a user-friendly graphical representation and a 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 parameterised classes of properties. Fig. 1 shows the representation of the *response* template $\Box(x \Rightarrow \Diamond y)$ in *Declare* and its possible instantiation in a process for renting apartments, where parameters x and y take the values *Plan final inspection* and *Execute final inspection.* This constraint means that every action *Plan final inspection* must eventually be followed by action *Execute final inspection.*

Since *Declare* models are focused on ruling out forbidden behaviour, *Declare* is very suitable for defining *compliance models* that are used for checking that the behaviour of a system complies certain regulations. The compliance model defines the rules related to a single instance of a process, and the expectation is that all the instances follow the model. For example, the constraint *after planning a final inspection, the inspection must be eventually executed* will be satisfied for a process instance trace if the activity "plan final inspection" is followed by "execute final inspection", or if "plan final inspection" does not happen at all. Note that this constraint is only informative for a company if "plan final inspection" can happen in some process instance; if it never happens, the constraint is still satisfied but in an uninteresting way, which is called *vacuous satisfaction.*

Vacuous satisfaction of a constraint might signal that the constraint from some reference compliance model is (1) irrelevant for a particular company (e.g., the company does not do final inspections at all), or (2) it might indicate some underspecification in the compliance model (e.g., the constraint saying that every process execution should contain the planning of the final inspection is missing). In case (1), the model is unnecessarily difficult for (new) company employees and they get inclined to disregard it as irrelevant. The company needs this constraint to be strengthened to the constraint saying that "plan final inspection" cannot occur. In case (2), the danger is even bigger, since the model does not capture the regulations correctly.

In this paper, we take the information about the executions of process instances captured in the event log (assuming that it is long enough [4] and thus captures enough information about the system behaviour), we check which constraints are vacuously satisfied on the log and we replace them with stronger constraints using our compliance patterns. We start from the existing results in the field of vacuity detection for single traces [2,6] and we extend the method

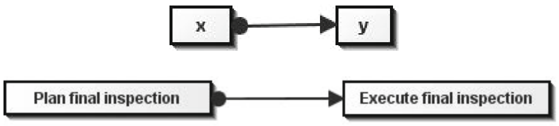

Fig. 1. Response template in *Declare* and its possible instantiation

of [6] in order to take into account the context of compliance checking where constraints are evaluated not just on single traces but on sets of traces coming from an event log.

We have evaluated our approach on an event log from a Dutch rental agency. Starting from an input compliance model defined by a domain expert we applied our approach and diagnosed the options for strengthening the compliance model so that the obtained model shows constraints relevant with respect to the log.

The remainder of the paper is structured as follows: we discuss related work in Sect. 2. In Sect. 3, we provide an informal introduction to the *Declare* language based on the *Declare* model for cancellation of a rental contract of an apartment rental company that we use as a running example. In Sect. 4, we propose compliance patterns for *Declare* constraints and in Sect. 5 we discuss our methodology for strengthening compliance models. In Sect. 6, we show an application of our approach to a real-life example. Section 7 concludes the paper.

2 Basic Concepts and Related Work

In Tab. 1, we briefly introduce the standard LTL operators and their (informal) semantics [16], which is used in *Declare* constraints.

In this paper, we start from the notions of *vacuity detection* and *interesting witness* first introduced in [2] for CTL* formulas. Since CTL* is a superset of LTL (in which we are interested in the context of *Declare*), we can apply these notions in our work. According to [2], a path π is an *interesting witness* for a formula φ if π satisfies φ non-vacuously, which means that every subformula ψ of φ affects the truth value of φ in π. In [2], the authors present an approach for vacuity detection for w-ACTL, a subset of Action Computational Tree Logic, which is, in turn, a subset of CTL. In [17], the authors present an approach for

Table 1. The LTL operators and their semantics

Operator	Semantics
$\bigcirc\varphi$	*φ holds in the next position of a path.*
$\Box\varphi$	*φ holds always in the subsequent positions of a path.*
$\Diamond\varphi$	*φ holds eventually (somewhere) in the subsequent positions of a path.*
$\varphi\mathcal{U}\psi$	*φ holds in a path at least until ψ holds. ψ must hold in the current or in a future position.*

vacuity detection in CTL formulas. They do not provide, however, an operative algorithm to be applied to LTL specifications.

In [6], 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 true. 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. These conditions 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 being a subformula of φ:

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

This approach was applied to *Declare* in [11] for vacuity detection in the context of process discovery. However, the algorithm introduced in [6] can generate different results for equivalent LTL formulas. Consider, for instance, the following equivalent formulas (corresponding to a *Declare alternate response* constraint):

$$\varphi = \Box(a \Rightarrow \Diamond b) \wedge \Box(a \Rightarrow \bigcirc((\neg a \mathcal{U} b) \vee \Box(\neg b))) \text{ , and}$$

$$\varphi' = \Box(a \Rightarrow \bigcirc(\neg a \mathcal{U} b)).$$

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

$$witness(\varphi) = \mathit{false},$$

$$witness(\varphi') = \varphi' \wedge \Diamond(\neg \bigcirc (\neg a \mathcal{U} b)) \wedge \Diamond(a) \wedge \Diamond(a \wedge \neg \bigcirc (b)).$$

In compliance models, LTL-based declarative languages like *Declare* are used to describe requirements to the process behaviour. 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 behaviour in an LTL-based process model.

Another issue in the approach proposed by [6] is that for two LTL formulas f and g, the composite formula $\varphi = f \vee g$ is *never non-vacuously true.* This is definitely counterintuitive, because one would expect that φ is non-vacuously true if f is non-vacuously true or g is non-vacuously true.

Furthermore, the notion of vacuous satisfaction, as introduced in [2,6], is designed for formulas that hold *on a given trace in an uninteresting way.* However, when applying (1) in the context of a log (a set of traces), we obtain, in some cases, conditions for vacuity detection that are too strong and difficult to satisfy.

For instance, if we apply (1) to $\varphi = \Box(\mathit{Agree\ on\ self\ made\ changes?} \Rightarrow \Diamond(\mathit{Plan\ final\ inspection} \vee \mathit{Adjust\ floor\ plan}))$, we obtain that $witness(\varphi)$ is:

$$\varphi \wedge \Diamond(\mathit{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathit{Plan\ final\ inspection})) \wedge$$
$$\Diamond(\mathit{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathit{Adjust\ floor\ plan})).$$

This formula is too strong in the context of a log, since we will not have in every trace *Agree on self made changes?* followed by both *Plan final inspection* and *Adjust floor plan*. In our approach, we will "weaken" this condition by requiring that each term of the conjunction must be valid, separately, on different traces. In addition, the original formula must be also always valid. This yields the following two formulas:

$$\varphi \wedge \Diamond(\mathit{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathit{Plan\ final\ inspection})), \text{ and}$$
$$\varphi \wedge \Diamond(\mathit{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathit{Adjust\ floor\ plan}))$$

each of which is expected to hold independently on some trace of the log to justify that the original formula is non-vacuously satisfied in the log.

3 Declare

Declare is characterised by a user-friendly graphical front-end and is based on a formal LTL back-end. These characteristics are crucial for two reasons. First of all, *Declare* is understandable for end-users and suitable to be used by stakeholders with different backgrounds. For instance, *Declare* has been already effectively applied in a project for maritime safety and security [9] where several project members did not have any formal background. Secondly, *Declare* has a formal semantics and *Declare* models are verifiable. This characteristic is important for the implementation of tools to check the compliance of process behaviour to *Declare* models (see, e.g., [10]).

A *Declare* model consists of a set of constraints which, in turn, are based on templates. Templates are parameterised classes of properties (a superset of the ones defined by Dwyer et al. in [3]) equipped with a graphical representation and a semantics specified through LTL formulas. Each *Declare* constraint inherits the graphical representation and semantics from its template. When a template is instantiated in a constraint, a template parameter is replaced by one or several activities. When two or more activities are used for one parameter, we say that this parameter branches and it is then substituted by a disjunction of branched activities in the LTL formula.

Declare constraints can be subdivided into four groups: *existence* (i.e., *existence*, *absence*, *exactly* and *init*), *relation* (i.e., *responded existence*, *co-existence*, *response*, *precedence*, *succession*, *alternate response*, *alternate precedence*, *alternate succession*, *chain response*, *chain precedence* and *chain succession*), *negative relation* (i.e., *not co-existence*, *not succession* and *not chain succession*), and *choice* (i.e., *choice* and *exclusive choice*). The LTL semantics for existence and relation constraints are listed in Tab. 2, Tab. 3 and Tab. 4 (first line for each constraint). For the full overview of the language we refer the reader to [13,15].

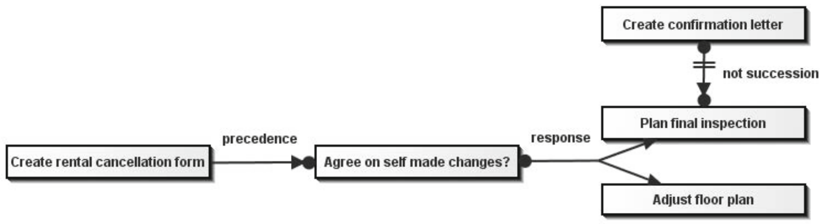
Create confirmation letter
not succession
Plan final inspection
precedence
response
Create rental cancellation form
Agree on self made changes?
Adjust floor plan

Fig. 2. An example of a *Declare* model

Fig. 2 shows a *Declare* model that describes a process for cancellation of a rental contract at a rental agency, which we use to explain the main characteristics of the language. The process in Fig. 2 involves five activities, depicted as rectangles (e.g., *Plan final inspection*), and three constraints, showed as connectors between the activities (e.g., *not succession*). In our example, prior to agreeing to any changes made by the tenant, the company must create a rental cancellation form. This form will be used in the activity *Agree on self made changes?* to specify whether the company agrees or disagrees with the self-made changes. This is indicated by the *precedence* constraint. After agreeing or disagreeing with the self-made changes, the company either plans a final inspection (to determine whether the tenant has reverted or mended her self-made changes), or adjusts the floor plan to reflect the current situation after the self-made changes. If they partially agree on the changes made by the tenant, it is possible to adjust the floor plan *and* plan a final inspection. All this is captured in the branched *response* constraint. Finally, the company cannot plan a final inspection after having created (and sent) a confirmation letter (stating that no problem was encountered), as indicated by the *not succession* constraint.

The response constraint in Fig. 2 is an example of a branched *response* constraint $\Box(x \Rightarrow \Diamond y)$, where parameter x is replaced by *Agree on self made changes?* and parameter y is branched on *Plan final inspection* and *Adjust floor plan*. This means that if *Agree on self made changes?* occurs in a trace, it must be eventually followed by *Plan final inspection* or *Adjust floor plan*, captured in LTL as $\Box(\textit{Agree on self made changes} \Rightarrow \Diamond(\textit{Plan final inspection} \vee \textit{Adjust floor plan}))$.

The other two constraints in our renting agency model, are captured in LTL as $\Box(\textit{Create confirmation letter} \Rightarrow (\neg\Diamond \textit{Plan final inspection}))$ (*not succession*), and $(\neg \textit{Agree on self made changes?}\ \mathcal{U}\ \textit{Create rental cancellation form}) \vee \Box(\neg \textit{Agree on self made changes?})$ (*precedence*).

The semantics of the whole model is determined by the conjunction of these formulas. Note that, when operating on business processes, we reason on finite traces. Therefore, we assume that the semantics of the *Declare* constraints is expressed in FLTL [8], a variant of LTL for finite traces.

4 Approach

In the remainder of the paper, we write $\mathcal{A}$ for the disjunction over the activities in $A = \{a_1, \cdots, a_n\}$, i.e., $\mathcal{A} = \bigvee_{a \in A} a$. Similarly, we write $\mathcal{B}$ for the disjunction $\bigvee_{b \in B} b$ over the activities in $B = \{b_1, \cdots, b_m\}$. We write W for an event log, $t \in W$ for a trace in W and we assume the activities to be atomic.

Similarly to the notion for vacuity detection captured by (1), we define a vacuity detection condition as follows:

Definition 1. *Given a (branched) Declare constraint φ, a* vacuity detection condition *of φ is a formula $\neg\varphi[\psi \leftarrow \bot]$ with ψ being a subformula of φ.*

In the context of compliance checking, we do not reason in terms of single traces but in terms of event logs that are composed of multiple traces. Therefore, as explained in Sect. 2, when applying (1) to a branched *Declare* constraint, instead of verifying the conjunction of all the vacuity detection conditions on every single trace, we adopt a more "permissive" approach. In particular, we require that for each vacuity detection condition, there exists a trace on which the condition holds.

According to [6], the algorithm described by (1) can be applied in a user-guided mode by limiting the evaluation of *witness*(φ) only to a subset of vacuity detection conditions. We choose these subsets differently for different *Declare* constraints by considering the vacuity detection conditions that give significant results from the point of view of each specific constraint. We have to use the user-guided mode because of the problems mentioned in Sect. 2.

As final output of our approach we want to obtain from a given compliance model and a log a more restrictive compliance model, where strengthening of the constraints is defined by the results of the vacuity check. The vacuity check can be done by applying a set of *compliance patterns*. First, through a compliance pattern, we check whether φ is satisfied everywhere in the log. Second, we check whether, for each vacuity detection condition $\neg\varphi[\psi \leftarrow \bot]$, there is a trace of the considered log where the condition is satisfied. Third, we check, for each constraint of the original model, whether a stronger constraint holds non-vacuously on every trace of the log. For this purpose, in Fig. 3, we define a hierarchy of *Declare* constraints where an arrow from a node x to a node y means that x implies y (x is stronger than y). The names of the constraints suggest their meaning and their semantics is defined later in this section. Note that RE^{-1} is used to denote that the sets of activities has been swapped. Therefore, from the hierarchy, *responded existence*(B, A) is weaker than *precedence*(A, B).

Based on these observations, we can define the compliance pattern of a *Declare* constraint:

Definition 2. *Given a log W and a (branched) Declare constraint φ, the* compliance pattern *of φ in W is a set composed of three conditions:*

1. *φ holds on every trace of W;*
2. *for each element of a selection of vacuity detection conditions of φ, there is a trace in W on which this element holds (user-guided application of (1));*
3. *no stronger constraint holds non-vacuously in W.*

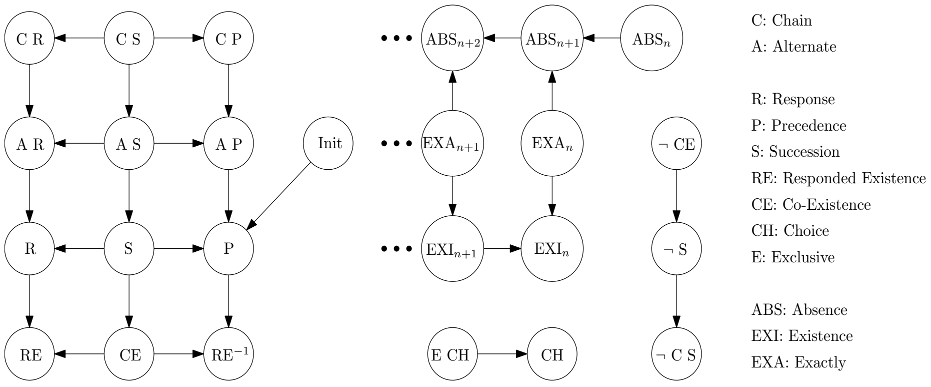

Fig. 3. The hierarchy of the *Declare* constraints

Furthermore, we define the notion of strong compliance as follows:

Definition 3. *Given a log W and a Declare constraint φ, W is* strongly compliant *to φ if all the conditions of the corresponding compliance pattern of φ are satisfied in conjunction on W.*

In the remaining subsections, we describe the compliance patterns of the *existence* and *relation* constraints. For the *negative relation* and *choice* constraints the application of the compliance patterns can be reduced to the evaluation of items 1 and 3 of Def. 2.

4.1 Existence Constraints

The compliance patterns for the existence constraints are listed in Tab. 2. For each constraint, the first line of the pattern shows the original LTL semantics that must hold for every trace in the log. For the *init* constraints, the additional condition is obtained by applying the approach for vacuity detection (1). When applying (1) on the $existence(nr, A)$, we replace $a \in A$ by false. Then, we obtain $\neg existence(nr, A[a \leftarrow false]) \equiv absence(nr, A \setminus \{a\})$.

Moreover, we want to ensure that in all these cases a stronger constraint does not hold (these conditions are not shown in the table for the sake of readability: they can be derived from the hierarchy in Fig. 3).

4.2 Relation Constraints

Our compliance patterns for the relation constraints are listed in Tab. 3 and Tab. 4. The conditions

$$\forall a \in A \; \exists t \in W : t \models \Diamond a, \text{ and } \forall b \in B \; \exists t \in W : t \models \Diamond b$$

must always be satisfied and we omit them in the tables. We also omit the conditions defining for each constraint that no stronger constraint holds non-vacuously in the log: they can be directly derived from the hierarchy in Fig. 3.

Table 2. Compliance patterns for existence constraints

Constraint	Pattern
existence$(1, A)$	$\forall t \in W : t \models \Diamond(\mathcal{A})$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$ $\forall a \in A : \exists t \in W : t \models \mathit{absence}(1, A \setminus \{a\})$
existence(nr, A)	$\forall t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(\mathit{existence}(nr-1, A)))$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$ $\forall a \in A : \exists t \in W : t \models \mathit{absence}(nr, A \setminus \{a\})$
absence(nr, A)	$\forall t \in W : t \models \neg \mathit{existence}(nr, A)$
exactly(nr, A)	$\forall t \in W : t \models \mathit{existence}(nr, A) \wedge$ $\mathit{absence}(nr+1, A)$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a)$
init(A)	$\forall t \in W : t \models \mathcal{A}$
	$\forall a \in A : \exists t \in W : t \models a$

Table 3. Compliance patterns for relation constraints without order

Constraint	Pattern
responded existence(A, B)	$\forall t \in W : t \models \Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B})$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A}) \wedge \Diamond(b)$
co-existence(A, B)	$\forall t \in W : t \models \mathit{responded\ existence}(A, B) \wedge$ $\mathit{responded\ existence}(B, A)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A}) \wedge \Diamond(b)$ $\forall a \in A : \exists t \in W : t \models \Diamond(\mathcal{B}) \wedge \Diamond(a)$

For each constraint, the first line of the pattern shows the original LTL semantics that must hold on every trace of the log. The additional conditions are obtained by applying (1) to the original semantics. Due to space restrictions we will only elaborate on the deduction of some compliance patterns.

Responded Existence. Applying (1) to *responded existence*(A, B), we replace $b \in B$ by false in the LTL formula $\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B})$. We obtain $\neg(\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B}[b \leftarrow \mathit{false}]))$. This formula is equivalent to $\Diamond(\mathcal{A}) \wedge \neg\Diamond(\mathcal{B}[b \leftarrow \mathit{false}])$. Combining this formula with *responded existence*(A, B) yields $\Diamond(\mathcal{A}) \wedge \Diamond(b)$. The condition of the compliance pattern of *responded existence*(A, B) is the combination of the conditions we obtain by replacing each $b \in B$ by false in the original formula.

Response. When we replace $b \in B$ in the LTL formula of *response*(A, B) by false, we obtain $\neg\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}[b \leftarrow \mathit{false}]))$. This is equivalent to $\Diamond(\mathcal{A} \wedge \neg\Diamond(\mathcal{B}[b \leftarrow \mathit{false}]))$. Considering that the original formula must be true, we can conclude that every $a \in A$ is not followed by any $b' \in B \setminus \{b\}$ is equivalent to every $a \in A$ is followed by b. This implies that $\Diamond(\mathcal{A} \wedge \Diamond(b))$. When we replace every $b \in B$ by false in the original formula, we have the condition of the pattern for *response*(A, B).

If we apply this pattern, for example, to *response* ({*Agree on self made changes?*}, {*Plan final inspection*, *Adjust floor plan*}), we obtain the following set of conditions that need to hold in the log:

Table 4. Compliance patterns for relation constraints with order

Constraint	Pattern
response(A, B)	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}))$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
precedence(A, B)	$\forall t \in W : t \models \neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
	$\exists t \in W : t \models \neg init(A)$
succession(A, B)	$\forall t \in W : t \models response(A, B) \wedge precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
	$\exists t \in W : t \models \neg init(A)$
alternate response(A, B)	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B})) \wedge \Box(\mathcal{A} \Rightarrow \bigcirc(\neg\mathcal{A}\mathcal{U}\mathcal{B}))$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
alternate precedence(A, B)	$\forall t \in W : t \models (\neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})) \wedge$ $\Box(\mathcal{B} \Rightarrow \bigcirc(\neg\mathcal{B}\mathcal{U}\mathcal{A} \vee \Box(\neg\mathcal{B})))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
alternate succession(A, B)	$\forall t \in W : t \models alt.\ response(A, B) \wedge alt.\ precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \Diamond(b))$
	$\forall a \in A : \exists t \in W : t \models (\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$
chain response(A, B)	$\forall t \in W : t \models \Box(\mathcal{A} \Rightarrow \bigcirc\mathcal{B})$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(b))$
chain precedence(A, B)	$\forall t \in W : t \models \Box(\bigcirc\mathcal{B} \Rightarrow \mathcal{A})$
	$\forall a \in A : \exists t \in W : t \models \Diamond(\bigcirc(\mathcal{B}) \wedge a)$
chain succession(A, B)	$\forall t \in W : t \models ch.\ response(A, B) \wedge ch.\ precedence(A, B)$
	$\forall b \in B : \exists t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(b))$
	$\forall a \in A : \exists t \in W : t \models \Diamond(a \wedge \bigcirc(\mathcal{B}))$

$\forall t \in W : t \models \Box(\textit{Agree on self made changes?} \Rightarrow$
$\Diamond(\textit{Plan final inspection} \vee \textit{Adjust floor plan}))$;
$\exists t \in W : t \models \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Plan final inspection}))$;
$\exists t \in W : t \models \Diamond(\textit{Agree on self made changes?} \wedge \Diamond(\textit{Adjust floor plan}))$.
Also, every constraint stronger than *response*({*Agree on self made changes?*}, {*Plan final inspection*, *Adjust floor plan*}) must not hold non-vacuously in the log.

Precedence. If we apply (1) to the LTL formula of *precedence*(A, B) and replace $a \in A$ by false, we obtain the condition $\neg((\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow \textit{false}])) \vee \Box(\neg\mathcal{B}))$ that is equivalent to $\neg(\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow \textit{false}])) \wedge \neg\Box(\neg\mathcal{B}))$. Similarly to the *response*(A, B), given that the original LTL formula holds, we have $(\neg\mathcal{B}\mathcal{U}a) \wedge \Diamond(\mathcal{B})$. When we replace every $b \in B$ in the original formula by true, we obtain the condition $\neg(\textit{false}\mathcal{U}\mathcal{A}) \vee \Box(\textit{false})$. This is equivalent to $\neg\mathcal{A}$, which means that there exists a trace where no $a \in A$ occurs at the first position.

Chain Response. Applying (1) to *chain response*(A, B), we replace in $\Box(\mathcal{A} \Rightarrow \bigcirc(\mathcal{B}))$ each $b \in B$ by false. We have $\neg\Box(\mathcal{A} \Rightarrow \bigcirc(\mathcal{B}[b \leftarrow \textit{false}]))$ that is equivalent to $\Diamond(\mathcal{A} \wedge \neg\bigcirc(\mathcal{B}[b \leftarrow \textit{false}]))$. Combining this formula with the original formula yields the condition $\Diamond(\mathcal{A} \wedge \bigcirc(b))$.

Take, for example, the constraint *chain response*({*Plan final inspection*}, {*Execute final inspection*, *Cancel final inspection*}). Applying this compliance pattern, we have:

$\forall t \in W : t \models \Box(\textit{Plan final inspection} \Rightarrow$
$\quad \bigcirc(\textit{Execute final inspection} \vee \textit{Cancel final inspection}))$;
$\exists t \in W : t \models \Diamond(\textit{Plan final inspection} \wedge \bigcirc(\textit{Execute final inspection}))$;
$\exists t \in W : t \models \Diamond(\textit{Plan final inspection} \wedge \bigcirc(\textit{Cancel final inspection}))$.

Moreover, *chain succession*({*Plan final inspection*}, {*Execute final inspection*, *Cancel final inspection*}) and *absence*(1, {*Plan final inspection*}) must not hold non-vacuously in the log.

5 Methodology

We present now a methodology for the application of the patterns from Sect. 4 in order to transform an existing compliance model into a strongly compliant one. We assume that the constraints of the existing model do hold on the log. Moreover, we check, for each activity in the model, whether it is present in the log. If not, we explicitly introduce an absence constraint on it.

Algorithm 1 lists the steps we execute to obtain a strongly compliant model M_{output} starting from a given compliant model M_{input}. We take a top-down approach, i.e., we start with the strongest constraints being candidates for strengthening and, according to the hierarchy defined in Fig. 3, we weaken them until we find a set of non-vacuously satisfied constraints (possibly the original constraint). When strengthening a constraint, we immediately remove branching on activities that do not occur in the log. For instance, if our model contains the constraint $\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}))$ and a $b \in B$ does not occur in the log, we strengthen the constraint to $\Box(\mathcal{A} \Rightarrow \Diamond(\mathcal{B}'))$ (where $B' = B \setminus \{b\}$). It cannot be the case that no $b \in B$ occurs in the log since this would mean that the constraint does not hold in the log. If b does not occur in the log, we add $absence(1, b)$ to the output model.

The algorithm relies on the following notion of a composed constraint:

Definition 4. *A* composed constraint φ *is a constraint that can be obtained by the conjunction of some other constraints, which we call components of* φ.

In Algorithm 1, we write C for the set of the components of constraint c; $C = \{c\}$ if c is not composed. For instance, for $c = \textit{chain succession}(A, B)$ we have $C = \{\textit{chain response}(A, B), \textit{chain precedence}(A, B)\}$, for $c = \textit{co-existence}(A, B)$ we have $C = \{\textit{responded existence}(A, B), \textit{responded existence}(B, A)\}$, and for $\varphi = \textit{alternate response}(A, B)$, $C = \{\textit{alternate response}(A, B)\}$.

For each constraint in the model, we first check whether it is the case that the constraint is of the type $\textit{precedence}(A, B)$ and $\textit{init}(A)$ holds non-vacuously. If so, we add $\textit{init}(A)$ to the output model. If the constraint is not of the type $\textit{precedence}(A, B)$ or $\textit{init}(A)$ does not hold non-vacuously, we check whether (a)

Algorithm 1. Transforming a compliant model to strongly compliant

```
Input: $M_{input}$ a model, $W$ a log
Output: $M_{output}$ a strongly compliant model
(1)  $M'_{input} \leftarrow$ an empty model
(2)  foreach Constraint $c$ in $M_{input}$
(3)      substitute $c$ by the strongest constraint w.r.t. the hierarchy and add it to $M'_{input}$
(4)  while $M'_{input}$ is not empty
(5)      $M_{temp} \leftarrow$ an empty model
(6)      foreach constraint $c$ in $M'_{input}$
(7)          if $c$ is precedence(A, B) and init(A) holds non-vacuously on $W$ then
(8)              add init(A) to $M_{output}$
(9)          else
(10)             if precedence(A, B) $\in C$ and init(A) holds non-vacuously on $W$ then
(11)                 $c_p \leftarrow$ precedence(A, B)
(12)                 add init(A) to $M_{output}$
(13)                 add all $c' \in C \setminus \{c_p\}$ which hold non-vacuously on $W$ to $M_{output}$ and add the remaining constraints in $C \setminus \{c_p\}$ to $M_{temp}$
(14)             else if each $c' \in C$ holds non-vacuously on $W$ then
(15)                 add $c$ to $M_{output}$
(16)             else if some $c' \in C$ hold non-vacuously then
(17)                 add all $c' \in C$ which hold non-vacuously on $W$ to $M_{output}$ and add the remaining constraints in $C$ to $M_{temp}$
(18)             else if no $c' \in C$ holds non-vacuously on $W$ then
(19)                 substitute $c$ by its immediate weaker notion and add it to $M_{temp}$
(20)     replace $M'_{input}$ by $M_{temp}$
(21) return $M_{output}$
```

precedence(A, B) is in C and *init*(A) holds non-vacuously, or (b) all constraints in C hold non-vacuously, or (c) a subset of the constraints in C holds non-vacuously, or (d) all constraints in C do not hold non-vacuously.

In the first case, we add the *init* and all non-vacuously satisfied constraints from C to the output model. The remaining constraints of C are added to the temporary model M_{temp} to be processed in the next iteration. In the second case, we add the constraint to the output model since in this case the constraint holds non-vacuously. In the third case, we add the subset of non-vacuously satisfied constraints in C to the output model and we add the remaining constraints in C to the temporary model to be processed in the next iteration. Consider, for instance, an *alternate succession* constraint where only the component *alternate response* is non-vacuously satisfied. In this case, we add the *alternate response* component to M_{output} and we keep the *alternate precedence* for future iterations.

In the fourth case, we add one of the weaker constraints (following the hierarchy defined in Fig. 3) to the temporary model.

When we check whether a constraint holds non-vacuously, we also remove vacuously satisfied branches of the constraint.

6 Case Study

We now present a small case study provided by a Dutch apartment rental agency in the form of an event log, recording process executions of a process for the cancellation of the rental contract by a tenant, and a compliance model defined by their domain expert (showed in Fig. 4). When the tenant gives a notice, the rental agency has to perform inspections to determine that the apartment is in a proper state. Based on these inspections, further actions might be needed.

Given an input model and a log, the question we want to pose is: *Does this compliance model correctly reflect the behaviour of the process represented in the log, assuming that the behaviour complies the model?* Note that we only want to facilitate the answering of this question for the domain expert. The final answer is up to the user, who can decide in which parts the strongly compliant model that our approach generates provides her with relevant information.

Starting from the model in Fig. 4, we apply the methodology from Sect. 5. First, we verify whether each activity in the model occurs at least once in the log. The activity *Create rental cancellation* never occurs in the log, so we add the constraint $absence(1, 7)$ to the output model (labelled with “0” in Fig. 5). Moreover, we replace all constraints by the strongest constraints with respect to the hierarchy introduced in Fig. 3. In particular, we replace the *precedence* constraint and all the *response* constraints by *chain succession* constraints. Moreover, we replace *not succession* by *not co-existence.*

After that, for all constraints, we check which of them hold non-vacuously on the log. To do this we use the LTL Checker plug-in of ProM[1]. The LTL Checker allows us to verify the validity of an LTL formula on a log. We use it to verify the validity of the conditions of a compliance pattern.

The *not co-existence* constraint holds on the log. This is enough to add this constraint to our output model. Indeed, this constraint is always non-vacuously true and there is no constraint stronger than *not co-existence.* All *chain succession* constraints added after the previous step do not hold: both the *chain response* component and the *chain precedence* component of each *chain succession* constraint do not hold. Therefore, we replace the *chain succession* constraints by *alternate succession* constraints. None of the *alternate succession* constraints hold, i.e., both the *alternate response* component and the *alternate precedence* component of each *alternate succession* constraint do not hold.

We replace then all *alternate succession* constraints by *succession* constraints. The *succession* constraints are also composed constraints. If we first consider *succession*(3, {4, 5, 6}), we need to have that *response*(3, {4, 5, 6}) and *precedence*(3, {4, 5, 6}) must hold non-vacuously. In the log, *response*(3, {4, 5, 6})

[1] `www.processmining.org`

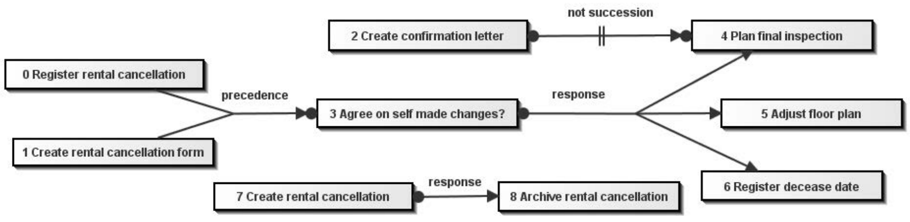
0 Register rental cancellation
1 Create rental cancellation form
precedence
2 Create confirmation letter
not succession
4 Plan final inspection
3 Agree on self made changes?
response
5 Adjust floor plan
6 Register decease date
7 Create rental cancellation
response
8 Archive rental cancellation

Fig. 4. Input model

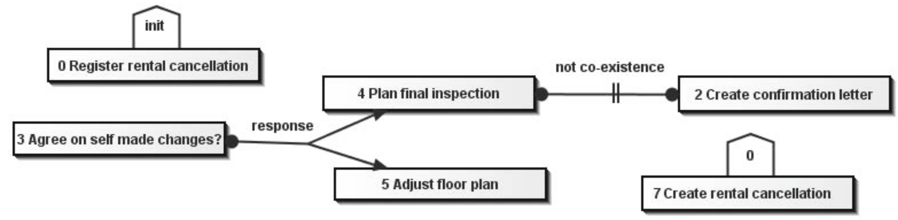
init
0 Register rental cancellation
4 Plan final inspection
not co-existence
2 Create confirmation letter
3 Agree on self made changes?
response
5 Adjust floor plan
0
7 Create rental cancellation

Fig. 5. Output model

Table 5. The compliance of the different conditions on the log

Constraint	Compliance pattern conditions	Valid
precedence({0, 1}, 2)	$\forall t \in W : t \models \neg 2\mathcal{U}(0 \vee 1) \vee \Box(\neg 2)$	✓
	$\exists t \in W : t \models \neg 2\mathcal{U}0 \wedge \Diamond(2)$	✓
	$\exists t \in W : t \models \neg 2\mathcal{U}1 \wedge \Diamond(2)$	✓
	$\exists t \in W : t \models \neg init(\{0, 1\})$	✗
	a stronger constraint should not hold non-vacuously	✓
not succession(2, 4)	$\forall t \in W : t \models \Box(2 \Rightarrow \neg\Diamond(4))$	✓
	$\neg\forall t \in W : t \models \neg(\Diamond 2 \wedge \Diamond 4)$	✗
response(3, {4, 5, 6})	$\forall t \in W : t \models \Box(3 \Rightarrow \Diamond(4 \vee 5 \vee 6))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(4))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(5))$	✓
	$\exists t \in W : t \models \Diamond(3 \wedge \Diamond(6))$	✗
	a stronger constraint should not hold non-vacuously	✓
response(7, 8)	$\forall t \in W : t \models \Box(7 \Rightarrow \Diamond(8))$	✓
	$\exists t \in W : t \models \Diamond(7 \wedge \Diamond(8))$	✗
	a stronger constraint should not hold non-vacuously	✗

holds non-vacuously if we remove the branch on activity 6, so we add *response*(3, {4, 5}) to the output model. Moreover, *precedence*(3, {4, 5, 6}) does not hold, so we add this constraint to the temporary model to verify it in the next iteration. We also split *succession*({0, 1}, 3) into *response*({0, 1}, 3) and *precedence*({0, 1}, 3). We have that the *init*(0) holds non-vacuously, so we add *init*(0) to the output model. Moreover, *response*({0, 1}, 3) does not hold and we add it to the temporary model.

We have now two constraints we want to verify: *precedence*(3, {4, 5, 6}) and *response*({0, 1}, 3). Both do not hold in our log. *precedence*(3, {4, 5, 6}) is weakened to a *responded existence*({4, 5, 6}, 3). *response*({0, 1}, 3) is weakened to a *responded existence*({0, 1}, 3) and both are checked. Both *responed existence* constraints do not and are removed from the model. The strongly compliant model we obtain is depicted in Fig. 5. Here, all constraints hold non-vacuously.

All the constraints from the input model and their compliance patterns are listed in Tab. 5. For each pattern we have indicated whether every single condition is valid on the log or not. The table shows that for each constraint in the original model a part of the pattern is not valid on the log. Based on the results in Tab. 5, each constraint of the original model must be modified to obtain the strongly compliant model depicted in Fig. 5. The advantages of the output model with respect to the input model are: (1) precision, the output model describes reality better than the input model, and (2) understandability, one only has to understand the relevant parts.

7 Conclusion

In this paper, we describe compliance patterns for strengthening constraints in compliance models specified in *Declare* in order to show which part of the behaviour is actually covered by the process executions recorded in the event log of the system, and which (parts of) constraints are vacuously satisfied. This approach can be used for configuring reference models towards the needs of a company. We have shown in the case study how we make use of our constraints hierarchy to achieve the best results.

Our approach can easily be extended for the use in situations when some log traces violate a compliance model in order to produce a weakened compliance model showing what part of the compliance regulations does hold in the company practice.

For the future work, we plan to introduce quantitative measurements for vacuity, which are interesting in the context of large logs. In this case, a strengthened model can show in which way *most* of the process executions satisfy the compliance model, and which part of the behaviour is rather exceptional for the system in question. The quantitative approach can also be useful for logs with noise.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, M.H.: Declarative workflows: Balancing between flexibility and support. Computer Science - Research and Development 23, 99–113 (2009)
2. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient Detection of Vacuity in Temporal Model Checking. Formal Methods in System Design 18, 141–163 (2001)

3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM (1999)
4. van Hee, K.M., Liu, Z., Sidorova, N.: Is my event log complete? - A probabilistic approach to process mining. In: Proceedings of RCIS 2011, pp. 1–7. IEEE (2011)
5. Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P.: On Enabling Data-Aware Compliance Checking of Business Process Models. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 332–346. Springer, Heidelberg (2010)
6. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. International Journal on Software Tools for Technology Transfer 4(2), 224–233 (2003)
7. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing Declarative Logic-Based Models from Labeled Traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007)
8. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The Glory of the Past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
9. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: Analyzing Vessel Behavior using Process Mining in the Poseidon book edited by Springer (to appear)
10. Maggi, F.M., Montali, M., Westergaard, M., van der Aalst, W.M.P.: Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 132–147. Springer, Heidelberg (2011)
11. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Proceedings of CIDM 2011, pp. 192–199. IEEE (2011)
12. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring Business Constraints with the Event Calculus. Technical Report DEIS-LIA-002-11, University of Bologna, Italy (2011)
13. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
14. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-Based Workflow Models: Change Made Easy. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 77–94. Springer, Heidelberg (2007)
15. Pesic, M., Schonenberg, M.H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: Proceedings of EDOC 2007, pp. 287–300. IEEE Computer Society (2007)
16. Pnueli, A.: The Temporal Logic of Programs. In: Proceedings of FOCS 1977, pp. 46–57. IEEE Computer Society (1977)
17. Purandare, M., Somenzi, F.: Vacuum Cleaning CTL Formulae. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 485–499. Springer, Heidelberg (2002)