Do My Constraints Constrain Enough? — Patterns for Strengthening Constraints in Declarative Compliance Models

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

Eindhoven University of Technology, The Netherlands.
d.m.m.schunselaar@student.tue.nl, {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 a system, recorded in an event log, can be checked on its compliance to the given model.
A known pitfall in specifying such a set of constraints is allowing for more behaviour than the intended one. Consider, for instance, a model with a constraint specifying that an `invoice` must be eventually followed by a `payment`: if `invoice` never occurs in the event log at all, the behaviour is considered as compliant, while it is vacuously-compliant. Is this the behaviour we intended to model? Is this the right idea of compliance we had in mind? Undoubtedly, only the process designer who modelled that process can answer these questions. In this paper, we provide the designer with patterns that, applied to the model, trigger posing the right questions and define options for strengthening the constraints. Our patterns are inspired by vacuity detection techniques working on a single trace. We take the log point of view instead to check whether the constraints of a compliance model constrain enough.
Keywords: Linear Temporal Logic, Declare, Vacuity detection, Compliance checking

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. Consider, for instance, a physician in a hospital, who needs a high level of flexibility to take

* 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.

into account individual characteristics of a patient. At the same time, there are some general regulations and guidelines she has to follow. In such cases, declarative process models are more effective than the imperative ones [1,10]. Instead of explicitly specifying all the possible sequences of activities in a process, 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 what is not forbidden is allowed. In this way, models offer flexibility and still remain compact.

Recent works have shown that declarative languages based on LTL (Linear Temporal Logic) [11] can be fruitfully applied in the context of process discovery [6] and compliance checking [5,7]. In [8,9], the authors introduce an LTL-based declarative process modelling language called *Declare*. *Declare* is 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. The example in Figure 1 shows the representation of the response template $\Box(A \Rightarrow \Diamond B)$ in *Declare* and its possible instantiation in a process for renting apartments, where parameters A and B 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.

Due to the focus ruling out the forbidden behaviour, *Declare* is very suitable for defining *compliance models* that are used for checking that the behaviour of a system (e.g., recorded in an event log of the system) complies certain regulations. The compliance model dictates the rules for the execution of 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, eventually the inspection must be executed* will be satisfied on a process trace, if the planning of the final inspection was followed by the execution of this inspection, or if a final inspection was not executed at all. Note that the execution of the final inspection might be not required in every process execution; however, when specifying such a constraint, an expert normally expects that the final inspection is planned in at least one execution. Therefore, it is important to make a difference between constraint satisfaction in the expected way: there is at least one inspection planned and it is then followed by the execution of this inspection, and *vacuous satisfaction*, when no inspection was ever planned at all, which often signals a problem in the system behaviour.

Vacuous satisfaction can also be related to a general tendency of the designers to underspecification. 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, which might be the constraint that every process execution contains the planning of the final inspection.

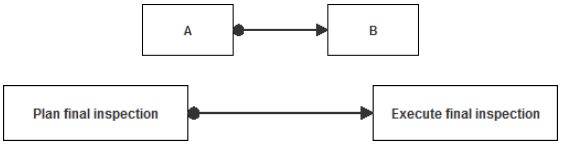

Fig. 1. Response template in *Declare* and its possible instantiation

In this paper, we start from the existing results in the field of vacuity detection [2,3] and provide a number of compliance patterns in order to help the process designer understand in which sense the system behaviour captured in an event log satisfies a compliance model. We achieve that by applying to the original model our patterns that allow to strengthen the compliance model "as much as possible" in order to show where the compliance requirements are vacuously satisfied. We extend the method of [3] 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 constraints of the compliance model with respect to the log.

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

2 Related Work

In Table 1, we briefly introduce the standard LTL operators and their (informal) semantics [11], which are used in *Declare* templates.

In this paper, we start from the notions of *vacuity* and *interesting witness* first introduced in [2] for CTL* formulas. Since CTL* is a superset of LTL (which we are interested in 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 presented an approach for vacuity detection for w-ACTL, a subset of Action Computational Tree Logic (ACTL), which is, in turn, a subset of CTL. In [12], the authors present an approach for vacuity detection in CTL formulas. They do not provide, however, an operative algorithm to be applied to LTL specifications.

Table 1. The LTL operators and their meaning.

Operator	Meaning
$\bigcirc\varphi$	*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.*

In [3], 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 $\neg\varphi[\psi \leftarrow \bot]$, for all the subformulas ψ of φ, 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:

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

This approach can in principle be applied to *Declare*. Indeed, in [6], it is applied for vacuity detection in the context of process discovery. However, the algorithm introduced in [3] can generate different results for equivalent LTL formulas. Consider, for instance, the following equivalent formulas (corresponding to the *Declare* alternate response template):

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

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

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

$$witness(\varphi) = false,$$

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

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 [3] 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, when considering vacuous satisfaction in the context of one single trace.

The notion of vacuous satisfaction, as introduced in [2,3], is designed for formulas that hold *on a given path in an uninteresting way*. In the context of a log (a set of traces), we would expect that $f \vee g$ is non-vacuously true if there is a trace in the log where f is non-vacuously true and there is a trace in the log where g is non-vacuously true. Consider, for example, the formula $\varphi = \Box(\mathsf{Agree\ on\ self\ made\ changes?} \Rightarrow \Diamond(\mathsf{Plan\ final\ inspection} \vee \mathsf{Adjust\ floor\ plan}))$.

When we apply (1) to this formula, we obtain that $witness(\varphi)$ is:

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

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:

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

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

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 [5,6,7] where several project members did not have any formal background. Secondly, being based on a formal semantics, *Declare* is verifiable. This characteristic is important for the implementation of tools to check the compliance of process behaviour to *Declare* models (see, e.g., [5]).

Figure 2 shows an example of a *Declare* model that describes a process for cancellation of a rental contract of a rental agency, which we use to explain the main characteristics of the language. The process involves five events, depicted as rectangles (e.g., Plan final inspection) and three constraints, shown as arcs between the events (e.g., not succession). In our example, prior to agreeing on any self made changes by the tenant, the company must create a rental cancellation form on which it can be specified whether the company agrees or disagrees with

the self made changes. This is indicated by the precedence constraint. After agreeing or disagreeing on 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. Also, if they partially agree on the changes made by the tenant, it is possible to both adjust the floor plan and plan a final inspection. This is indicated by the branched response, which says that if Agree on self made changes? occurs, Plan final inspection or Adjust floor plan have to occur after. Finally, the company cannot plan a final inspection after having created a confirmation letter (stating that no problem was encountered), as indicated by the not succession between the two events Create confirmation letter and Plan final inspection.

Declare provides templates that can be subdivided into four groups: *existence*, *relation*, *negative relation*, and *choice*. For the full overview of *Declare* templates we refer the reader to [8,9]. When being instantiated in a model, a template parameter in a *Declare* constraint is replaced by one or several activities. When two or more activities are used for one parameter, we say that this parameter branches. The response constraint in Figure 2 is an example of a branched response constraint $\Box(A \Rightarrow \Diamond B)$, where parameter A is replaced by Agree on self made changes? and parameter B 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. In case of branching, the parameter is replaced (a) by a multiple arcs to all branched activities in the graphical representation and (b) by a disjunction of branched activities in the LTL formula. The LTL semantics for the example above is:

$$\Box(X \Rightarrow \Diamond(Y \vee Z))$$

where X is Agree on self made changes? and Y and Z are Plan final inspection and Adjust floor plan respectively.

Each individual *Declare* constraint can be written as an LTL formula talking about the connected events. Using CL, CF, P, Ag and Ad to respectively denote Create confirmation letter, Create rental cancellation form, Plan final inspection, Agree on self made changes? and Adjust floor plan events in Figure 2, we obtain $\mathsf{CF} \Rightarrow (\neg\Diamond\mathsf{P})$ for the not succession constraint, $(\Diamond\mathsf{Ag}) \Rightarrow (\neg\mathsf{Ag}\ \mathcal{U}\ \mathsf{CL})$ for the precedence constraint, and $\mathsf{Ag} \Rightarrow \Diamond(\mathsf{P}\vee\mathsf{Ad})$ for the (branched) response constraint in the renting agency model.

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 [4], a variant of LTL for finite traces.

4 Approach

In the remainder of the paper, we use the following notation: for a set $A = \{a_1, \cdots, a_n\}$ of formulas a_i, we write $\mathcal{A}$ for the disjunction over the elements of

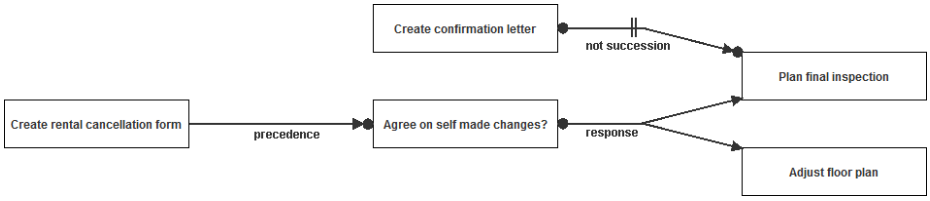

Fig. 2. An example of a *Declare* model

A, i.e., $\mathcal{A} = \bigvee_{a \in A} a$. Similarly, we write $\mathcal{B}$ for the disjunction $\bigvee_{b \in B} b$ over the elements of $B = \{b_1, \cdots, b_m\}$. We also indicate with W an event log and with $t \in W$ a trace in W. Moreover, we assume that the activities we consider are atomic.

From the algorithm for vacuity detection described 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 paths but in terms of event logs that are composed of multiple traces. Therefore, 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 [3], the algorithm described by (1) can be applied in a user-guided mode by limiting the evaluation of $witness(\varphi)$ only to a subset of vacuity detection conditions. We choose these subsets differently for different *Declare* templates by considering the vacuity detection conditions that give significant results from the point of view of each specific template.

As final output of our approach we want to adjust a given model incorporating the result of the vacuity check. If there is a vacuity detection condition $\neg\varphi[\psi \leftarrow \bot]$ that is not satisfied on any trace in the considered log, we conclude that part ψ of the model is not relevant for the entire log. To reflect that in the compliance model we, therefore, remove that part. In most cases, this part is a particular branch of the original constraint.

Considering that our goal is to strengthen the original model "as much as possible" towards the behaviour presented in an event log, we define a hierarchy of templates as depicted in Figure 3. We use the following abbreviations for the *Declare* templates: "RE" responded existence, "CE" co-existence, "R" response, "P" precedence, "S" succession, "AR" alternate response, "AP" alternate precedence, "AS" alternate succession, "CR" chain response, "CP" chain precedence, "CS" chain succession,"I" is the init template, implying the precedence one; "ABS", "EXA" and "EXI" stand for absence, exactly and existence respectively; "NCE", "NS" and "NCS" stand for not co-existence, not succession and not chain succession respectively, and, finally, "CHO" and "ECHO" denote choice

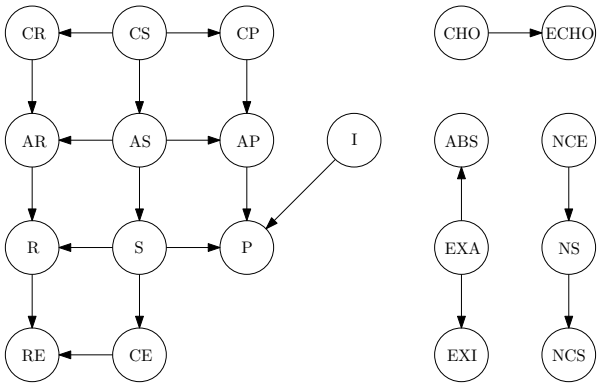

Fig. 3. The hierarchy of the different templates

and exclusive choice respectively. The names of the templates suggest their meaning, and their semantics is defined later in this section. An arrow from a node x to a node y means that x implies y.

In this way, we consider each constraint of the original model and, if a stronger (according to this hierarchy) constraint holds non-vacuously on every trace of the log, we report that it is possible to strengthen the original model by replacing this constraint by the stronger one.

Considering the aforementioned 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 four conditions:*

1. *all the activities involved in φ occur in W;*
2. *φ holds on every trace of W;*
3. *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));*
4. *no stronger constraint holds non-vacuously.*

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* templates. We do not show the *negative relation* and *choice* templates for the sake of brevity. Moreover, for the latter two templates (1) does not give significant results and the application of the compliance patterns can be reduced to the evaluation of items 1, 2 and 4 of Definition 2.

4.1 Existence templates

Our compliance patterns for the existence templates are listed in Table 2. Since the condition

$$\forall a \in A\ \exists t \in W : t \models \Diamond a$$

must be always valid, we omit it in the table. For each template, the first line of the pattern shows the original LTL semantics that must hold for every trace in the log. For the init template, the additional condition is obtained by applying the approach for vacuity detection (1). On the other hand, when applying (1) to the existence, absence, and exactly templates, we obtain that for every $a \in A$ there has to be a trace such that a occurs at least nr times in that trace. We do not consider these conditions in the proposed version of the compliance patterns because they are too strong and difficult to apply to real cases. However, we want to ensure that in all these cases a stronger template does not hold.

In particular, if we want to ensure that existence(nr, A) is strongly compliant, we need to specify a condition guaranteeing that existence(nr, A) cannot be replaced by existence$(nr + 1, A)$. Moreover, we also need a second condition to ensure that existence(nr, A) cannot be replaced by exactly(nr, A).

Similarly, if we want to ensure that absence(nr, A) is strongly compliant, we must verify a condition specifying that absence(nr, A) cannot be replaced by absence$(nr - 1, A)$. A second condition guarantees that absence(nr, A) cannot be replaced by exactly$(nr - 1, A)$.

Finally, when exactly(nr, A) holds, it is always strongly compliant, since there is no template stronger than exactly(nr, A).

Assume that we have the following constraint in our model: absence(3, {Plan final inspection, Adjust floor plan}), i.e., the amount of times we can plan a final inspection or adjust a floor plan is at most 3. Applying the compliance patterns to this constraint yields the following conditions that need to be valid on the log:

$$\forall t \in W : t \models \neg\text{existence}(3, \{\text{Plan final inspection}, \text{Adjust floor plan}\})$$
$$\exists t \in W : t \models \text{existence}(2, \{\text{Plan final inspection}, \text{Adjust floor plan}\})$$
$$\exists t \in W : t \models \text{absence}(2, \{\text{Plan final inspection}, \text{Adjust floor plan}\})$$

4.2 Relation templates

Our compliance patterns for the relation templates are listed in Tables 3 and 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 be always satisfied and we omit them in the tables. We also omit in the tables the conditions that ensure that, for each constraint, no stronger constraint

Table 2. Compliance patterns for existence templates.

Template	Pattern
$existence(1,\ A)$	$\forall t \in W : t \models \Diamond(\mathcal{A})$
	$\exists t \in W : t \models \neg existence(2, A)$
	$\exists t \in W : t \models existence(2, A)$
$existence(nr,\ A)$	$\forall t \in W : t \models \Diamond(\mathcal{A} \wedge \bigcirc(existence(nr-1, A)))$
	$\exists t \in W : t \models \neg existence(nr+1, A)$
	$\exists t \in W : t \models existence(nr+1, A)$
$absence(nr,\ A)$	$\forall t \in W : t \models \neg existence(nr, A)$
	$\exists t \in W : t \models existence(nr-1, A)$
	$\exists t \in W : t \models absence(nr-1, A)$
$exactly(nr,\ A)$	$\forall t \in W : t \models existence(nr, A) \wedge$
	$absence(nr+1, A)$
	$true$
$init(A)$	$\forall t \in W : t \models \mathcal{A}$
	$\forall a \in A : \exists t \in W : t \models a$

must hold non-vacuously on the log: they can be directly derived from the hierarchy in Figure 3. For each template, 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.

Table 3. Compliance patterns for relation templates without order.

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

Responded existence Applying (1) to $\mathsf{responded\ existence}(A, B)$, we replace a $b \in B$ by false in the LTL formula $\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B})$ (Note that b is in the scope of an even number of negations). We obtain $\neg(\Diamond(\mathcal{A}) \Rightarrow \Diamond(\mathcal{B}[b \leftarrow false]))$. This formula is equivalent to $\Diamond(\mathcal{A}) \wedge \neg\Diamond(\mathcal{B}[b \leftarrow false])$. Combining this formula with the original formula yields

$$\Diamond(\mathcal{A}) \wedge \Diamond(b).$$

The condition of the compliance pattern of $\mathsf{responded\ existence}(A, B)$ is the combination of the conditions we obtain by replacing each $b \in B$ by false in the original formula.

Table 4. Compliance patterns for relation templates with order.

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

Response When we replace a $b \in B$ in the LTL formula of $\mathsf{response}(A, B)$ 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 $\mathsf{response}(A, B)$.

If we apply, for example, this pattern to $\mathsf{response}(\{\mathsf{Agree\ on\ self\ made\ changes?}\}, \{\mathsf{Plan\ final\ inspection}, \mathsf{Adjust\ floor\ plan}\})$, we obtain the following set of conditions that need to hold in the log:
$\forall t \in W : t \models \Box(\mathsf{Agree\ on\ self\ made\ changes?} \Rightarrow$
$\Diamond(\mathsf{Plan\ final\ inspection} \vee \mathsf{Adjust\ floor\ plan}))$;
$\exists t \in W : t \models \Diamond(\mathsf{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathsf{Plan\ final\ inspection}))$;
$\exists t \in W : t \models \Diamond(\mathsf{Agree\ on\ self\ made\ changes?} \wedge \Diamond(\mathsf{Adjust\ floor\ plan}))$.
In addition, Agree on self made changes?, Plan final inspection and Adjust floor plan must occur at least in one trace. 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 $\mathsf{precedence}(A, B)$ and replace $a \in A$ by false, we obtain the condition $\neg((\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow \mathit{false}])) \vee \Box(\neg\mathcal{B}))$ that is equivalent to $\neg(\neg\mathcal{B}\mathcal{U}(\mathcal{A}[a \leftarrow \mathit{false}])) \wedge \neg\Box(\neg\mathcal{B}))$. Similarly to the $\mathsf{response}(A, B)$, 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 $\neg(\mathit{false}\mathcal{U}\mathcal{A}) \vee \Box(\mathit{false})$. This is equivalent to $\neg\mathcal{A}$, which means that there exists a trace where no $a \in A$ occurs at the first position.

Applying this pattern to precedence({Register rental cancellation, Create rental cancellation form}, {Agree on self made changes?}) yields the following set of conditions that need to hold in the log:
$\forall t \in W : t \models (\neg$Agree on self made changes?$\mathcal{U}($Register rental cancellation$\vee$
Create rental cancellation form$)) \vee \Box(\neg$Agree on self made changes?$)$;
$\exists t \in W : t \models (\neg$Agree on self made changes?$\mathcal{U}$Register rental cancellation$) \wedge$
$\Diamond($Agree on self made changes?$)$;
$\exists t \in W : t \models (\neg$Agree on self made changes?$\mathcal{U}$Create rental cancellation form$) \wedge$
$\Diamond($Agree on self made changes?$)$;
$\exists t \in W : t \models \neg\mathsf{init}($Register rental cancellation $\vee$ Create rental cancellation form$)$.
Activities Register rental cancellation, Create rental cancellation form and Agree on self made changes? must occur at least once in the log. In addition, in every trace in the log, each constraint stronger than precedence({Register rental cancellation, Create rental cancellation form}, {Agree on self made changes?}) must not hold non-vacuously.

Alternate response Like in [3], we say that a conjunction is non-vacuously true if the conjuncts are non-vacuously true. Therefore, considering that the LTL semantics for alternate response(A, B) is response$(A, B) \wedge \Box(\mathcal{A} \Rightarrow \bigcirc(\text{precedence}(B, A)))$, we can use, as the first condition of the compliance pattern of alternate response(A, B), the condition derived for response(A, B). We then need to generate a condition to guarantee that the remaining part of the formula is non-vacuously true. Replacing in $\Box(\mathcal{A} \Rightarrow \bigcirc(\text{precedence}(B, A)))$ each $b \in B$ by false, we obtain $\neg(\Box\mathcal{A} \Rightarrow \bigcirc(\neg\mathcal{A}\mathcal{U}\mathcal{B}[b \leftarrow false]))$. This is equivalent to $\Diamond(\mathcal{A} \wedge \neg \bigcirc (\neg\mathcal{A}\mathcal{U}\mathcal{B}[b \leftarrow \mathit{false}]))$. Combining this formula with the original formula yields

$$\Diamond(\mathcal{A} \wedge \bigcirc(\neg\mathcal{A}\mathcal{U}b)).$$

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 $\neg\Box(\mathcal{A} \Rightarrow \bigcirc(\mathcal{B}[b \leftarrow \mathit{false}]))$ that is equivalent to $\Diamond(\mathcal{A} \wedge \neg \bigcirc (\mathcal{B}[b \leftarrow \mathit{false}]))$. 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:
$\forall t \in W : t \models \Box(\text{Plan final inspection} \Rightarrow$
$\bigcirc(\text{Execute final inspection} \vee \text{Cancel final inspection}))$;
$\exists t \in W : t \models \Diamond(\text{Plan final inspection} \wedge \bigcirc(\text{Execute final inspection}))$;
$\exists t \in W : t \models \Diamond(\text{Plan final inspection} \wedge \bigcirc(\text{Cancel final inspection}))$.
Moreover, Plan final inspection, Execute final inspection and Cancel final inspection must occur at least in one trace and chain succession({Plan final inspection}, {Execute final inspection, Cancel final inspection}) must not hold non-vacuously in the log.

5 Methodology

We present now a methodology for the application of the patterns from Section 4 in order to transforming an existing compliance model into a strongly compliant one.

Algorithm 1 lists the steps needed to be executed to obtain a strongly compliant model M_{output} starting from a given compliant model M_{input}. A constraint is strongly compliant if there are no stronger constraints that are non-vacuously true. Therefore, we perform a top-down approach, i.e., we start with the strongest constraints being candidates for strengthening according to the hierarchy defined in Figure 3 and weaken them until we find a set of non-vacuously satisfied constraints. The first step of the algorithm replaces every constraint in M_{input} by the strongest possible constraint with respect to the hierarchy as defined in Figure 3. When strengthening a constraint, we immediately remove branches on activities

Algorithm 1: Transforming a compliant model to strongly compliant
STRENGTHEN(M_{input})
Input: M_{input} a model
Output: M_{output} a strongly compliant model

```
(1)  foreach Constraint c in M_input
(2)      substitute c by the strongest constraint w.r.t. the hierarchy and add it to M'_input
(3)  while M'_input is not empty
(4)      M_temp ← an empty model
(5)      foreach constraint c in M'_input
(6)          if c is precedence(A, B) and init(A) holds non-vacuously then
(7)              add init(A) to M_output
(8)          else
(9)              if c_p = precedence(A, B) ∈ C and init(A) holds non-vacuously then
(10)                 add init(A) to M_output
(11)                 add all c' ∈ C \ {c_p} which hold non-vacuously to M_output and add the remaining constraints in C \ {c_p} to M_temp
(12)             else if each c' ∈ C holds non-vacuously then
(13)                 add c to M_output
(14)             else if some c' ∈ C hold non-vacuously then
(15)                 add all c' ∈ C which hold non-vacuously to M_output and add the remaining constraints in C to M_temp
(16)             else if no c' ∈ C holds non-vacuously then
(17)                 substitute c by its weaker notion and add it to M_temp
(18)     replace M'_input by M_temp
(19) return M_output
```

that do not occur in the log. If all the branches of a branched parameter have been removed, we remove the constraint from the 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 $c = \mathsf{chain\ succession}(A, B)$ we have $C = \{\mathsf{chain\ response}(A, B), \mathsf{chain\ precedence}(A, B)\}$, for $c = \mathsf{co\text{-}existence}(A, B)$ we have $C = \{\mathsf{responded\ existence}(A, B), \mathsf{responded\ existence}(B, A)\}$, and for $\varphi = \mathsf{alternate\ response}(A, B)$, $C = \{\mathsf{alternate\ response}(A, B)\}$.

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

init(A) does not hold non-vacuously, we check whether (a) 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 the weaker constraint (following the hierarchy defined in Figure 3) to the temporary model.

When we check whether a constraint holds non-vacuously, we also remove vacuously satisfied branches. We also remove a constraint from the model if we have to weaken it but a weaker constraint does not exist. For instance, for a precedence constraint according to the hierarchy in Figure 3 a weaker constraint does not exist.

6 Case study

We present now a case study provided by a Dutch apartment rental agency in the form of a event log recording process executions of a process for the cancellation of the rental contract by a tenant. After the tenant chooses to give 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.

We start from an input compliance model defined by a domain expert (depicted in Figure 4). To increase readability, we have added identifiers to refer to the different activities.

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 Figure 4, we apply the methodology from Section 5.

We first replace all constraints by the strongest constraints with respect to the hierarchy introduced in Figure 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. We also verify that each activity occurs at least once in the log. The activity Create rental cancellation

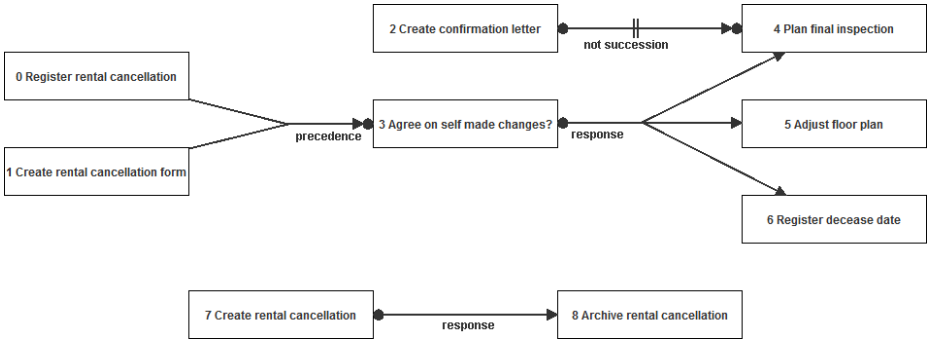

Fig. 4. Input model

never occurs in the log, so we can remove the constraint response(7, 8) from the model.

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. The other constraints do not hold, so we replace them by the composed constraints alternate succession. All alternate succession constraints added after the previous step do not hold: both the alternate response component and the alternate precedence component of each alternate succession constraint do not hold. Therefore we replace the 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. On the log, response(3, {4, 5, 6}) 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. However, precedence(3, {4, 5, 6}) cannot be weakened any further and is removed from the model. On the other hand, response({0, 1}, 3) is weakened to a responded existence({0, 1}, 3) and verified. Also responded existence({0, 1}, 3) does not hold and is removed from the model.

[1] www.processmining.org

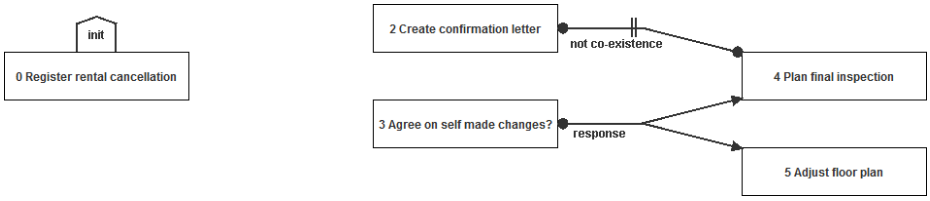

Fig. 5. Output model

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

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

The strongly compliant model we obtain is depicted in Figure 5. Here, all constraints hold non-vacuously.

All the constraints from the input model and their compliance patterns are listed in Table 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. For this reason, each constraint of the original model must be modified to obtain the strongly compliant model depicted in Figure 5.

7 Conclusion

In this paper we describe 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. We have shown on the case study that we do need to make use of the constraints hierarchy we defined to achieve the best results.

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.

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. Form. Methods Syst. Des. 18, 141–163 (2001)
3. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. International Journal on Software Tools for Technology Transfer 4(2), 224–233 (2003)
4. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The Glory of the Past. In: Proc. of Logic of Programs. pp. 196–218. Springer (1985)
5. 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: Proc. of BPM (2011)
6. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: 2011 IEEE Symposium on Computational Intelligence and Data Mining (2011)
7. Montali, M., Maggi, F.M., Chesani, F., Mello, P., van der Aalst, W.M.P.: Monitoring Business Constraints with the Event Calculus. Tech. Rep. DEIS-LIA-002-11, University of Bologna (Italy) (2011)
8. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops, Lecture Notes in Computer Science, vol. 4103, pp. 169–180. Springer (2006)
9. Pesic, M., Schonenberg, M.H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, 2007, EDOC 2007 (2007)
10. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: Proc. of the 2007 OTM Confederated international conference On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I. Lecture Notes in Computer Science, vol. 4803. Springer (2007)
11. Pnueli, A.: The temporal logic of programs. Foundations of Computer Science, Annual IEEE Symposium on 0, 46–57 (1977)
12. Purandare, M., Somenzi, F.: Vacuum cleaning CTL formulae. In: Proceedings of the 14th International Conference on Computer Aided Verification. pp. 485–499. Springer-Verlag (2002)