University of Groningen

Efficient conditional compliance checking of business process models

Groefsema, Heerko; van Beest, Nick; Armas-Cervantes, Abel

Published in:
Computers in Industry

DOI:
[10.1016/j.compind.2019.103181](https://doi.org/10.1016/j.compind.2019.103181)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2020

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Groefsema, H., van Beest, N., & Armas-Cervantes, A. (2020). Efficient conditional compliance checking of business process models. *Computers in Industry*, 115, [103181].
<https://doi.org/10.1016/j.compind.2019.103181>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Efficient conditional compliance checking of business process models

H. Groefsema^{a,*}, N.R.T.P. van Beest^{b,*}, A. Armas-Cervantes^c

^a Distributed Systems Group, University of Groningen, The Netherlands

^b Software Systems Group, Data61 | CSIRO, Brisbane, Australia

^c School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Article history:

Received 12 May 2019

Received in revised form 8 November 2019

Accepted 6 December 2019

Available online 17 December 2019

Keywords:

Business process models

Formal verification

Conditional compliance

Data perspective

Temporal logic

ABSTRACT

When checking compliance of business processes against a set of business rules or regulations, the ability to handle and verify conditions in both the model and the rules is essential. Existing design-time verification approaches, however, either completely lack support for the verification of conditions or propose costly verification methods that also consider the full data perspective. This paper proposes a novel light-weight verification method, which is preferable over expensive approaches that include the data perspective when considering structural properties of a business process model. This novel approach generates partial models that capture only relevant execution states to the conditions under investigation. The resulting model can be verified using existing model checking techniques. The computation of such partial models fully abstracts conditions from the full models and specifications, thus avoiding the analysis of the full data perspective. The proposed method is complete with respect to the analyzed execution paths, while significantly reducing the state space complexity by pruning unreachable states given the conditions under investigation. This approach offers the ability to check if a process is compliant with rules and regulations on a much more fine-grained level, and it enables a more precise formulation of the conditions that should and should not hold in the processes. The approach is particularly useful in dynamic environments where processes are constantly changing and efficient conditional compliance checking is a necessity. The approach – implemented in Java and publicly available – is evaluated in terms of performance and practicability, and tested over both synthetic datasets and a real-life case from the Australian telecommunications sector.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Laws and regulations affect the way businesses are run on a daily basis. With constantly changing laws, companies are forced to increase the flexibility and agility of their business processes. Consequently, the information systems supporting those business processes continue to evolve, such that it becomes increasingly difficult to ensure their compliance to laws and regulations. As a result, a need for automated compliance checking of business processes has become evident.

Compliance checking of business processes aims to prove whether or not a business process adheres to a set of imposed rules [1,2]. These rules can stem from laws, regulations, internal business rules, or even as part of a specification describing different variants of the process [3,4]. Compliance checking can be

supported at various stages of the business process lifecycle [5]. For example, many existing approaches enable compliance checking either during enactment using monitoring techniques, or during analysis using mining techniques. In this context, we focus on *design-time verification* where *preventative* solutions for possible non-compliance can be implemented. This is distinct from after-the-fact approaches (such as the aforementioned monitoring and analysis techniques) where costly rollbacks could be required to undo any faulty non-compliant execution.

The ability to handle and verify conditional branches (e.g. “manager approval is required if the loan amount is larger than \$50,000”) is an essential requirement for compliance verification. Existing design-time verification approaches, however, either completely lack support for verification over conditions given business process models or require inclusion of the data perspective (i.e. taking into account the entire domain of each data attribute), which can quickly cause state space explosion problems.

This paper proposes a novel method that fully abstracts conditions from a model and its specifications. This is achieved by generating partial models, based on the overlap and contradictions

* Corresponding authors.

E-mail addresses: h.groefsema@rug.nl (H. Groefsema), nick.vanbeest@data61.csiro.au (N.R.T.P. van Beest).

between the specified conditions. The result is a light-weight verification method over the structure of the business process that is preferable over expensive verification methods that include the data perspective, as the internal representations are simplified significantly while still enabling verification only over the paths that are reachable under the given condition. We prove that when the partial models are combined, they contain the complete set of possible paths as represented in the full model given its condition, while simultaneously reducing its state space complexity in the relevant cases.

A high-level overview of the approach is shown graphically in Fig. 1 below. First, we take as input a Petri net with conditional branches and a set of conditional temporal logic specifications (1). Next, we generate a set of partial models, specific to each condition in the specification (2). Finally, we verify compliance over every partial model (3).

As such, the contribution of the paper is fourfold: (i) it demonstrates that the full data perspective is not required for the verification of structural properties over conditional branches, (ii) it shows how to formally abstract conditions from the models and specifications, (iii) it shows how to obtain a formally verifiable partial model to conditions, and (iv) it proves that partial models preserve the same reachable states given their condition, while at the same time reducing the state space complexity in the relevant cases where the condition affects the number of reachable states.

The approach presented in this paper can be used by organizations to check if their business process models are compliant with established rules and regulations on a much more fine-grained level when compared to compliance checking without support for conditions. As such, it enables organizations to formulate the rules that should hold in their processes in a much more precise way compared to some existing approaches that ignore the data perspective, while ensuring that each of their processes is fully compliant. The presented approach is particularly useful in dynamic environments where processes are changing frequently and efficient data-oriented conditional compliance checking is a necessity and frequently required, such that performance gains are significant.

The remainder of the paper is structured as follows. First, Section 2 discusses related work and positions our work within the existing state of the art. Section 3 introduces the preliminary paradigms used for both business process models and verification (Step 1 in Fig. 1). Subsequently, Section 4 describes how to obtain partial models (Step 2 in Fig. 1) and how to interpret specifications on those models (Step 3 in Fig. 1). Section 5 describes a real life case from the Australian telecommunications sector, which – together with a set of synthetic process models – is used in Section 6 to evaluate the proposed approach with respect to complexity, performance, and practicability. Finally, the paper is concluded in Section 7.

2. Related work

Compliance verification aims to prove or disprove whether a business process adheres to a set of rules, like laws, regulations, business rules or internal policies [2]. In this way, compliance verification does not aim to prove the correctness of the business process itself (like e.g. [6–8]), but merely whether it adheres to a set of rules. Compliance can be verified at different stages of the business process lifecycle, including *design-time*, *enactment*, and *analysis* [5]. At design-time, compliance can be verified given a *business process model* or the *implementation* that is based on that business process model. Given an implementation, compliance can be verified using monitoring techniques during enactment, or using mining techniques during analysis. When an implementation is verified on the

extent to which it adheres to a business process model, the process is called *conformance checking*. Conformance checking, however, does *not* prove compliance unless the business process model used has been proven compliant first. For example, an implementation can be conformant to the model but not compliant (i.e. it violates some regulations). This would imply that the model is not compliant either. On the other hand, an implementation can be compliant but not conformant to the model. In this case, the implementation allows for behavior that is not allowed in the model, but that additional behavior is not violating any regulations it is subject to.

Formal compliance verification aims to prove or disprove compliance using formal methods (e.g. model checking). Within this context, we focus on *design-time compliance verification* using *formal methods* given *business process models*, as it is the only stage of the business process lifecycle at which costly erroneous process implementations and enactments can be prevented. An overview of the related work is given in Fig. 2, where it has been divided into *process-centric* and *data-centric* domains and whether conditions are considered. For a detailed comparison and evaluation of the proposed model translation and translations of other methods *without considering conditions*, the reader is referred to [2].

Several formal compliance verification approaches have been introduced that check compliance of process-centric business process models, like e.g. [9,10,12,16,2]. These approaches focus on the control-flow aspect of compliance and use a variety of logics, including deontic logics [9,10], linear-time temporal logic [12,16], or computation tree logic [2]. The complexity of these approaches is combinatorial for parallel behavior of activities and linear otherwise [2]. Some approaches directly encode the business process into the modeling language of a model checker, which may result in either a large amount of overhead resulting in state explosion [12,16] or major omissions in the formal properties of the underlying modeling (e.g. with respect to OR-joins) [21]. Conditions in process-centric business process models are usually modeled as boolean expressions using simple textual labels on outgoing flows of branching design elements (i.e., gateways). While the ability to handle such conditions over different branches is natural given the implementation of a business process, the data perspective is not included using conventional business process modeling languages. As a result, these methods fail to consider conditions.

Formal design-time compliance verification techniques are often restricted to acyclic models (see for example [11,13–15]). Arbitrary cycles are a very powerful feature of business process model design and they naturally occur in real-life business processes as well. In [2], cyclic models are supported, while the notion of fairness is used to allow correct verification over cycles. When allowing for cycles, however, the necessity of support for verification using conditions becomes apparent. The order of activities and specific paths through arbitrary cycles depend on the conditions on the branches and simply introducing a fairness condition may oversimplify the result. As such, the ability to handle and verify conditional branches may strongly improve the expressiveness of the specification and, as a consequence, precision of the result.

Verification of data-centric business processes and systems is introduced in [19,20]. These approaches, however, focus on *behavioral properties* of the data perspective of the systems. The complexity of these approaches is exponential to the size of the domain of the data. Instead, we focus on *structural properties* of the commonly used process-centric control-flow perspective, while taking conditions into account.

To allow verification of the control flow using data, the data perspective is introduced to process-centric approaches in [17,18]. In [17] the business process model is simply annotated with the data perspective. Some approaches verify processes on business rules

Fig. 1. Overview of the approach.

	Process-centric	Data-centric
No Conditions	e.g. [9, 10, 11, 12, 13, 14, 15, 16, 2]	
Conditions	This work and [17, 18]	e.g. [19, 20]

Fig. 2. Overview of related work.

[22,23], by taking the entire data state space into account. However, the complexity of these methods is that of the formal verification approach times that of the data-centric approach. As a result, each of these methods have a high risk of state space explosion.

In [18] data abstractions are obtained for both the model and the compliance rules to reduce the amount of possible states when checking compliance on data and conditions. The model checking is subsequently performed on the full model with the abstractions. As a result, the data abstractions can be used to verify *behavioral properties* of the business process model including conditional paths. The complexity of this method is that of the formal verification approach times the complexity of the data abstractions. Although there is a practical reduction in complexity using the data abstractions with respect to using the full data perspective, there is no theoretical reduction.

Instead, we pair formulas with data conditions to reduce the model itself given the data and conditions without requiring the full data perspective. That is, the state space of the model is reduced per data condition to only contain those paths that correspond to that condition. As a result, the models obtained can be used to verify *structural properties* of the business process model given the conditions over certain paths in the model. That is, instead of verifying whether an activity is followed by another activity in each process execution when some data condition holds, we verify whether an activity is followed by another activity in the paths of the business process model that adhere to that data condition. By verifying structural properties of the business process model, we can then assert certain behavioral properties. The proposed method shows that the full data perspective is not required to verify such structural properties. The complexity of the proposed method is that of the number of different conditions used times that of the formal verification approach under a practical reduction given each data condition.

3. Preliminaries

We now first introduce the preliminary paradigms used for modeling business process models and verifying those business process models subject to a specified set of rules, as required by Step 1 in Fig. 1.

There are different notations for modeling business processes, mostly using an intuitive graphical representation. In the context of this work, we use *Petri nets*, a well-known modeling tool for concurrent processes, for which a rich body of theory and tools to verify their properties have been defined. A Petri net is a directed bipartite graph consisting of places, transitions, and arcs between transition-place pairs. The activities in a business process are represented as transitions in a Petri net, while the execution states are represented as combinations of places, which are token containers. A Petri net is formally defined as follows [24,25]:

Definition 1 (*Petri net*). A tuple $\langle P, T, A \rangle$ is a *Petri net*, where:

- P is a set of *places*,
- T is a set of *transitions*, such that $P \cap T = \emptyset$, and
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs.

A Petri net execution state $M : P \rightarrow \mathbb{N}_0$, also known as *net marking*, is a function that associates places with natural numbers (viz., place tokens). A *marked net* $N = \langle P, T, A, M_0 \rangle$ is a Petri net $\langle P, T, A \rangle$ with an *initial marking* M_0 .

Places and transitions are referred to as *nodes*. The *preset* of a node is denoted by $\bullet y = \{x \in P \cup T \mid (x, y) \in A\}$, while the *postset* of a node is denoted by $y \bullet = \{z \in P \cup T \mid (y, z) \in A\}$. Using pre- and postsets, the execution semantics of Petri nets can be defined in terms of markings. If $\forall p \in \bullet t : M(p) > 0$, t is said to be *enabled* and may fire. The *firing* of t , denoted by $M \xrightarrow{t} M'$, leads to a new marking M' ,

Fig. 3. A simple Petri net.

with $M'(p) = M(p) - 1$ if $p \in \bullet t \setminus t\bullet$, $M'(p) = M(p) + 1$ if $p \in t\bullet \setminus \bullet t$, and $M'(p) = M(p)$ otherwise. The marking M_n is said to be reachable from M if there exists a sequence of transition firings $\sigma = t_1 t_2 \dots t_n$ such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} M_n$. A marking M of a net is n -safe if $M(p) \leq n$ for all $p \in P$. A Petri net N is n -safe if all its reachable markings are n -safe. In the remainder of this paper, we restrict ourselves to 1-safe nets. Fig. 3 shows a simple Petri net.

Formal verification entails checking if a model representing a system conforms to a set of specifications. One approach to formal verification is model checking, where systems are represented as labeled transition systems. In order to use model checking techniques over a Petri net, we use state-based labeled transition systems called *Kripke structures* [26] to represent the execution state space (i.e., markings) of a Petri net.

Definition 2 (Kripke structure). Let AP be a set of atomic propositions. A Kripke structure K over AP is a quadruple $K = \langle S, S_0, R, L \rangle$, where:

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation such that it is left-total, meaning that for each $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$, and
- $L: S \rightarrow 2^{AP}$ is a labeling function over the set of atomic propositions.

Kripke structures are often used to interpret temporal logics, formalisms that are able to reason over execution paths within models. Computation Tree Logic (CTL) [27] is one of the most notable temporal logics that is commonly used in formal verification. CTL is a branching-time temporal logic, which is especially useful when considering *structural* properties (in contrast to *behavioral* properties) of the model [28], over the different branching constructs in the process models.

Definition 3 (CTL syntax). The language of well-formed CTL formulas is generated by the following grammar, assuming $p \in AP$:

$$\phi ::= \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \\ AX\phi \mid EX\phi \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi \mid A[\phi \cup \phi] \mid E[\phi \cup \phi]$$

CTL is equipped with four *temporal operators*:

- $X\phi$ Nexttime: ϕ has to hold at the next state.
- $G\phi$ Globally: ϕ has to hold on every state on the entire subsequent path.
- $F\phi$ Future: ϕ eventually has to hold in a future state.
- $[\phi \cup \phi']$ Until: ϕ has to hold until ϕ' , which holds at a state on the path from the current state or the current state itself.

Each temporal operator ψ in CTL is paired with an operator ϕ over paths. Operators over paths specify if some or all branches possess properties starting at the current state. CTL defines the following operators over paths:

- $A\psi$ All: ψ holds on all paths flowing from the current state.
- $E\psi$ Exists: ψ holds on at least one path flowing from the current state.

The semantics of CTL are defined on a Kripke structure, which are formally defined in Appendix A. The following section presents a method for using CTL to verify structural requirements on the control flow of a business process over partial representations of Kripke structures. The partial representations of the Kripke structures are computed with respect to conditions associated to the CTL formulas.

4. Verification over guards

Following the basic definitions of the model, we can now describe the procedure to obtain a set of partial models from the full model as visualized by Step 2 in Fig. 1. The presented method requires two elements as input, (1) a process model annotated with conditional paths, and (2) a set of conditional CTL rules (pairs of CTL rules and conditions). The conditions in the model and the conditions in the conditional CTL rules are formally defined as follows:

Definition 4 (Condition). A condition C is a quantifier-free first-order predicate, using variables $\bar{x} = x_1, \dots, x_n$, with $x_i \in X$ for $0 < i \leq n$, the fragment of logical connectives $\{\vee, \wedge, \neg\}$, and the fragment of binary relations $\{<, \leq, =, \geq, >, \neq\}$.

4.1. Guards

Business process models can contain conditions on certain paths, which only allow execution of those respective paths when that condition is satisfied. For example, a loan may only be approved if the customer is creditworthy. As such, the activity “Approve loan” would only be allowed to execute if “credit check” returns “ok”.

In order to represent conditional paths in a Petri net, we require so-called *guard conditions* to associate transitions to boolean expressions. A condition is represented as a guard, which is associated to a transition t and has to evaluate to *true* before t is allowed to fire (irrespective of whether the preset of that transition contains enough tokens). [29] shows how guards can be associated over the transitions of a Petri net. A guard over a transition is formally defined as follows:

Definition 5 (Guard condition). Given a marked net $N = \langle P, T, A, M_0 \rangle$ and a set of variables X , a guard condition G_t is a condition defined over a transition $t \in T$, using variables $\bar{x} = x_1, \dots, x_n$, with $x_i \in X$ for $0 < i \leq n$.

We only require guard conditions to map to a first-order expression, and do not require the full data perspective to verify structural properties of the business process model given guard conditions. Given a transition $t \in T$ and a set of values for the variables in $\bar{x}$, the guard condition G_t evaluates to a boolean value. Hence, transition t is only considered enabled if $\forall p \in \bullet t: M(p) > 0$ and its guard G_t evaluates to *true*. For example, Fig. 4 illustrates a Petri net annotated with guard conditions. Finally, trivial guards (e.g., $G_a = [\text{true}]$ of transition a) have been omitted for readability purposes.

In order to determine the truth values of a (guard) condition, a specific set of values needs to be assigned to the variables in the respective condition such that it evaluates to *true*. To determine such values, we use the first-order interpretation function I that associates a value to every variable and to every condition a set of tuples of values. Intuitively, the set of tuples associated to a condition are the values for which the condition is true, where C defines the set of conditions. An interpretation satisfies a condition if the

Fig. 4. Example of a Petri net with guards.

set of values assigned by the interpretation to the variables is one of the tuples that satisfies the condition:

$$\begin{aligned}
 I & : X \rightarrow V \\
 I & : C \rightarrow 2^V \\
 I \models G_t \wedge C & \Leftrightarrow \langle I(x_1), \dots, I(x_n) \rangle \in I(G_t) \cap I(C) \\
 I \models \bigwedge_{t \in Y} G_t \wedge C & \Leftrightarrow \langle I(x_1), \dots, I(x_n) \rangle \in \bigcap_{t \in Y} I(G_t) \cap I(C)
 \end{aligned}$$

In Fig. 4, for example, the interpretation $I(G_b) = \{0, 1, 2, \dots\}$ given the guard $[x \geq 0]$ on b . For $I(x) = 2$, $I(x) \in I(G_b)$, whereas for $I(x) = -1$, $I(x) \notin I(G_b)$. As such, the interpretation function can be used to obtain the set of values such that a given set of guards evaluate to *true*. This is particularly useful when evaluating whether certain paths can be executed in a Petri net, by allowing to identify the potential contradiction between different guards, as shown in detail below.

4.2. Computation of conditional graphs

When verifying business processes with guards, certain rules to be verified might only hold under certain conditions. For example, the rule “manager approval is required if the loan amount is larger than \$50,000” implies that the manager must always give his approval under the *condition* that the loan amount exceeds \$50,000. As such, CTL rules can be paired with a condition, to form a so-called *conditional CTL rule*. A conditional CTL rule is a tuple $\langle C, \phi \rangle$ consisting of a condition C and a CTL formula ϕ . The example above would then be represented as $\langle [\text{loan amount} > 50000], \text{AF 'manager approval'} \rangle$. Note that for a rule that must hold *at all times* (i.e. not subject to a condition), the tuple would be $\langle [\text{true}], \phi \rangle$.

The condition is then used to find sets of transitions that can fire given (i) a marking, (ii) the guards of the transitions in the Petri net, and (iii) the condition C . Given these sets of transitions for each reachable marking, a *conditional graph* is obtained that contains only those paths that are reachable under the given condition C . A conditional graph is a Kripke structure computed from the markings of a Petri net that are reachable under a condition C , and has the capability of representing the *parallel* execution of tasks and the *next* execution of a task in a parallel branch [2,1]. This Kripke structure is then used to verify the CTL formula ϕ contained in the conditional CTL rule.

To obtain a conditional graph from a marked net, a set of states (including the initial state), a set of relations between states, and

a set of atomic propositions to label states must be created. To create states, the different sets of transitions that can be enabled simultaneously given a condition at a marking M must be obtained first. Given that a transition $t \in T$ of a Petri net is enabled when $\forall p \in \bullet t : M(p) > 0$ holds and its guard does not contradict with the condition, then the set of all enabled transitions given a marking M and a condition C can be defined as the set:

$$\mathfrak{X}(M, C) = \{t \mid t \in T, \forall p \in \bullet t, \exists I : M(p) > 0 \wedge I \models G_t \wedge C\}$$

Consider, for example, the marking $M_i = 1p_2 + 1p_3$ given the Petri net depicted in Fig. 4 and the condition $C_i = [y < 5]$. That is, consider the case where there exists one token at the place p_2 plus one token at the place p_3 for $y < 5$. Since p_2 contains a token, both c and τ_1 are enabled given the available tokens. In addition, the condition $y < 5$ does not directly contradict the guards of either transition. That is, there exist values of y such that $y < 0 < 5$ for τ_1 and values $0 \leq y < 5$ for c . Similarly, since p_3 contains a token, both d and τ_2 are enabled given the available tokens. However, the condition $y < 5$ directly contradicts the guard of τ_2 , which is therefore not enabled. In other words, the set of enabled transitions for the marking $M_i = 1p_2 + 1p_3$ and the condition $C_i = [y < 5]$ contains c, d , and τ_1 : $\mathfrak{X}(M_i, C_i) = \{c, d, \tau_1\}$.

The sets of enabled transitions, however, only contain those transitions that are enabled individually, and do not determine whether any of the transitions could be enabled and fired simultaneously considering the available tokens and guards of each transition. Given marking M_i in the example above, for instance, c and τ_1 are both enabled but cannot fire both. That is, they are not *enabled in parallel*. Given a marking M , condition C , and interpretation I providing a set of values for variables of guards, a set of transitions Y is enabled in parallel if the following boolean expression holds:

$$\begin{aligned}
 E(M, C, I, Y) & \stackrel{\text{def}}{=} (I \models \bigwedge_{t \in Y} G_t \wedge C) \wedge (\forall t \in Y, \forall p \in \bullet t : M(p) \\
 & \geq |p \bullet \cap Y|)
 \end{aligned}$$

Using the power set of the set of enabled transitions $\mathcal{P}(\mathfrak{X}(M, C))$ and the function $E(M, C, I, Y)$ to determine whether a set is enabled in parallel, all the different sets of parallel enabled transitions at marking M are obtained as the set:

$$\begin{aligned}
 \mathfrak{Y}_{\text{par}}(M, C) & = \{Y \mid Y \in \mathcal{P}(\mathfrak{X}(M, C)) : (\exists I : E(M, C, I, Y) \\
 & \wedge (\nexists t \in \mathfrak{X}(M, C) : E(M, C, I, Y \cup \{t\})))\}
 \end{aligned}$$

That is, each set of transitions Y is considered if and only if:

- all transitions t in Y are enabled such that they can fire simultaneously (i.e. the presets of all t in Y do not overlap), and
- if there exists an interpretation I for the variables in X that satisfy the combination of guards $\bigwedge_{t \in Y} G_t$ given the condition C , and
- if the set is maximal to the interpretation I , such that no additional transition can be enabled simultaneously given I .

For example, consider again the marking is $M_i = 1p_2 + 1p_3$ given the Petri net depicted in Fig. 4 and the condition $C_i = [y < 5]$. Even though both c and τ_1 are enabled for the marking $M_i = 1p_2 + 1p_3$, the place p_2 only contains a single token while the guards $G_c = [y \geq 0]$ and $G_{\tau_1} = [y < 0]$ also contradict. As a result, only one of the enabled transitions may actually fire. This fact is reflected in the set of parallel enabled transitions $\mathcal{Y}_{par}(M_i, C_i) = \{\{c, d\}, \{\tau_1, d\}\}$, which contains two sets of transitions that could fire simultaneously. That is, c can fire simultaneously with d , and τ_1 can fire simultaneously with d . As mentioned above, c and τ_1 cannot fire simultaneously and are, as such, not included as a set in $\mathcal{Y}_{par}(M_i, C_i)$.

Using $\mathcal{Y}_{par}(M, C)$, we compute the Kripke structure of a marked net by creating a state for each reachable marking M_i and for each set of transitions that can fire concurrently at M_i under condition of C . Next, each concurrent transition is fired individually to find possible next states [2,1]. We say a marking M_n is *reachable under condition of C* from M_0 if there exists a sequence of transition firings $\sigma = t_0 t_1 \dots t_{n-1}$ such that $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \dots M_{n-1} \xrightarrow{t_{n-1}} M_n$ and the transitions $\forall i : t_i \in \mathcal{Y}(M_i, C)$ are enabled under condition of C . Observe that $I \models G_{t_0} \wedge \dots \wedge G_{t_{n-1}} \wedge C$ is not required to hold, because of the possible variable manipulation outside of the boundaries specified by C . A Kripke structure computed from a marked net given a condition C is referred to as a *conditional graph* and defined formally below.

Definition 6 (Conditional graph). Let AP be a set of atomic propositions representing markings and transitions of a marked net $N = \langle P, T, A, M_0 \rangle$. A conditional graph is a tuple $\langle C, K \rangle$ where C is a condition and $K = \langle S, S_0, R, L \rangle$ is a Kripke structure over AP obtained from the marked net N with reachable markings $M_0, \dots, M_n$ under condition of C , using:

- $AP = \{M_0, \dots, M_n\} \cup \{t \in Y \mid Y \in \mathcal{Y}_{par}(M_0, C) \cup \dots \cup \mathcal{Y}_{par}(M_n, C)\}$,
- $S = \{s_i^Y \mid Y \in \mathcal{Y}_{par}(M_i, C) : 0 \leq i \leq n\}$,
- $S_0 = \{s_0^Y \mid Y \in \mathcal{Y}_{par}(M_0, C)\}$,
- $L(s_i^Y) = \{M_i\} \cup Y$, and
- $R = \{(s_i^Y, s_j) \mid t \in Y \wedge M_i \in L(s_i^Y) \wedge M_j \in L(s_j) \wedge M_i \xrightarrow{t} M_j \wedge L(s_i^Y) \setminus \{M_i, t\} \subseteq L(s_j)\} \cup \{(s_i^Y, s_i^Y) \mid Y = \emptyset\}$.¹

For instance, consider the Petri net depicted in Fig. 4 with initial marking $M_0 = 1p_{start}$ and the condition $[x = 1]$. Although all markings in the net are reachable given that condition, the Kripke structure K in the conditional graph $\langle [x = 1], K \rangle$ (depicted in Fig. 5) does not contain a state with f . This is a result of the function $\mathcal{Y}_{par}(M, [x = 1])$, which does not return the transition f in this case, because $G_f = [x < 0]$ contradicts the condition $[x = 1]$. Note that while the variable x is being limited, the variable y remains free in this example.

Although a conditional graph of a marked net requires a condition C , it is possible to also obtain the full graph by simply using the

condition true . In this case, guards are unaffected by the condition and all possible paths are included in the graph. The resulting graph $\langle \text{true}, K \rangle$ is known as the transition graph of the marked net [2,1].

Lemma 1 (State space complexity). The state space of the Kripke structure $K^C = \langle S^C, S_0^C, R^C, L^C \rangle$ of a conditional graph $\langle C, K^C \rangle$ of a marked net N is always smaller than, or equal to, the state space of the full Kripke structure $\langle \text{true}, K \rangle$ of the same Petri net.

Proof. Both K^C and K are constructed from the set of reachable markings in N , $\mathcal{M}^C$ and $\mathcal{M}$ respectively. Any marking M_i is included if it is reachable under condition of C or true , respectively. Given that it is only possible to remove markings which are not reachable under condition of C , it follows that $\mathcal{M}^C \subseteq \mathcal{M}$.

For each of the reachable markings, states are included for each set of transitions in $\mathcal{Y}_{par}(M_i, C)$ or $\mathcal{Y}_{par}(M_i, \text{true})$ respectively. Since it is only possible to remove transitions that are not enabled under condition of C , it holds that each set in $\mathcal{Y}_{par}(M_i, C)$ is either included in $\mathcal{Y}_{par}(M_i, \text{true})$, is a subset of a set in $\mathcal{Y}_{par}(M_i, \text{true})$, or is excluded for having become a subset of, or equivalent to, another set in $\mathcal{Y}_{par}(M_i, C)$. Thus, it holds that $|\mathcal{Y}_{par}(M_i, C)| \leq |\mathcal{Y}_{par}(M_i, \text{true})|$ and $\forall Y^C \in \mathcal{Y}_{par}(M_i, C), \exists Y \in \mathcal{Y}_{par}(M_i, \text{true}) : Y^C \subseteq Y$. Because $\mathcal{M}^C \subseteq \mathcal{M}$ and $|\mathcal{Y}_{par}(M_i, C)| \leq |\mathcal{Y}_{par}(M_i, \text{true})|$, it follows that $S^C \subseteq S$ and $S_0^C \subseteq S_0$, and because $\forall Y^C \in \mathcal{Y}_{par}(M_i, C), \exists Y \in \mathcal{Y}_{par}(M_i, \text{true}) : Y^C \subseteq Y$, it holds that $\forall s \in S^C : L^C(s) \subseteq L(s)$.

Relations (s_i^Y, s_j) are included for $M_i \xrightarrow{t} M_j$ and a state s_i^Y , with $M_i \in L(s_i^Y)$, $t \in Y$, and $M_j \in L(s_j)$. In addition, reflexive relations (s_i^Y, s_i^Y) are included for states without a future state $Y = \emptyset$. Since $\mathcal{M}^C \subseteq \mathcal{M}$ and $\forall Y^C \in \mathcal{Y}_{par}(M_i, C), \exists Y \in \mathcal{Y}_{par}(M_i, \text{true}) : Y^C \subseteq Y$, it follows that $\forall (s_i^Y, s_j^{Y'}) \in R^C : \exists (s_i^Y, s_j^{Y'}) \in R \vee s_i^Y = s_j^{Y'}$ and $\forall (s_i^Y, s_i^Y) \in R^C : \exists (s_i^Y, s_i^Y) \in R \vee (\exists s_j \in S : (s_i^Y, s_j) \in R)$. As a result, it holds that $|R^C| \leq |R|$. $\square$

Although one could consider K^C as being a sub-graph of K (i.e., $S^C \subseteq S$, $S_0^C \subseteq S_0$, and $\forall s \in S^C : L^C(s) \subseteq L(s)$) it does not hold that the relations are a subset of the full Kripke structure, i.e., $R^C \not\subseteq R$, but only that $|R^C| \leq |R|$. This is due to the fact that the set of additional reflexive relations introduced at each end state to fulfill the left-total requirement of the set of relations in a Kripke structure (cf. Definition 2).

4.3. Evaluation of CTL rules

Finally, the temporal logic formulas can be evaluated over the conditional graphs (Step 3 in Fig. 1). Although it is possible to embed the conditions within the temporal logic formulas instead, this would increase the state space when considering data (as it would require to analyze the full data perspective) or only allow evaluation over the exact guards as atomic propositions. Instead, the proposed method decomposes the evaluation into two steps, the computation of the conditional graphs with respect to the conditions associated to the CTL rules, and the check of the CTL rules over the conditional graphs that are Kripke structures restricted by those conditions. A CTL formula ϕ holds under condition of C if it holds for all initial states of the Kripke structure K of the conditional graph $\langle C, K \rangle$. This can be defined formally as follows:

Definition 7 (Semantics of conditional CTL). Given a conditional Kripke structure $\langle C, K \rangle$, a state $s_i \in S$ of that Kripke structure, and a conditional CTL formula $\langle C', \phi \rangle$, we say that $\langle C, K \rangle, s_i \models \langle C', \phi \rangle$ holds if $K, s_i \models \phi$ and $C \equiv C'$. We also write $\langle C, K \rangle \models \langle C, \phi \rangle$ iff $\langle C, K \rangle, s_0 \models \langle C, \phi \rangle$ holds for all $s_0 \in S_0$.

A conditional CTL formula $\langle C, \phi \rangle$ can be read as *when C holds within the process flow, ϕ must also hold*. For example, the conditional CTL formula $\langle [x = 1], \text{AG}(a \Rightarrow \text{AF } e) \rangle$ can be verified for the Petri net depicted in Fig. 4 with initial marking $M_0 = 1p_{start}$ by using the

¹ Although Definition 6 uses elements from the definition itself to define R (i.e. the labeling function L), this is merely done to produce a more concise and readable definition.

Fig. 5. Kripke structure of the Petri net with condition $[x = 1]$. Fig. 6. Kripke structure of the Petri net with condition $[y = 10]$.

Kripke structure K of the conditional graph $\langle [x = 1], K \rangle$, see Fig. 5. The CTL formula $\text{AG}(a \Rightarrow \text{AF } e)$ holds for this Kripke structure.

Because we are verifying structural compliance of a process (as opposed to behavioral compliance), the condition C is treated as being invariant throughout the process. Although we allow for possible variable manipulation throughout the conditional graph, and allow therefore a marking to be reachable on a path with consecutive contradicting guards, this variable manipulation always remains within the bounds specified by C . In this way, a marking on a path which contains a guard that contradicts C is not reachable because variable manipulation is bounded by C .

As a result, possible blocking runs can be detected by C , due to consecutive contradicting guards. Consequently, bad process design through guards can be detected. For example, consider the verification of the conditional CTL formula $\langle [y = 10], \text{AF } g \rangle$ against the Petri net depicted in Fig. 4. The resulting Kripke structure K of the conditional graph $\langle [y = 10], K \rangle$ is illustrated in Fig. 6. Observe that neither the execution of g nor the execution of h is represented in K because their guards $G_g = [y < 10]$ and $G_h = [y > 10]$ do not consider the case where $y = 10$. Therefore, the CTL formula $\text{AF } g$ is evaluated to *false* in K (Fig. 6) since the markings where g occurred is not reachable (blocking run).

The computational complexity of verifying a formula ϕ on a Kripke structure K is $O(|K| |\phi|)$. Since the state space of K^C of a conditional graph $\langle C, K^C \rangle$ is equal or less than that of the state space of the Kripke structure K of the full graph $\langle \text{true}, K \rangle$, see Lemma 1, there is no theoretical reduction in the complexity of verification of a formula. This is true because the condition C may have no effect on any of the guards included in the Petri net. In practical settings, however, one would only consider conditions with an effect on guards, since only those cases provide useful results. Therefore, in such settings, $|K^C| < |K|$, and the complexity of verification would therefore be decreased.

Theorem 1 (Reachability). *Given a condition C , every state $s \in S$ with a marking reachable under condition of C of the Kripke structure $K = \langle S, S_0, R, L \rangle$ of the full transition graph $\langle \text{true}, K \rangle$ is also reachable in the Kripke structure $K^C = \langle S^C, S_0^C, R^C, L^C \rangle$ of the conditional graph $\langle C, K^C \rangle$.*

Proof. Suppose there exists a state $s_n^Y \in S$ with a marking M_n that is reachable under a condition C and $Y \in \mathcal{Y}_{\text{par}}(M_n, C)$, then it holds that $M_n \in \mathcal{M}^C$. Since $M_n \in \mathcal{M}^C$, there must

exist a sequence of transition firings $\sigma = t_0, t_1 \dots t_{n-1}$ such that $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \dots M_{n-1} \xrightarrow{t_{n-1}} M_n$, for which the transitions $\forall i : t_i \in \mathcal{Y}(M_i, C)$ are enabled. Then, it holds that $M_i \in \mathcal{M}^C, s_i^{Y'} \in S^C : M_i \in L(s_i^{Y'}) \wedge t_i \in Y$, and $(s_i^{Y'}, s_{i+1}) \in R^C$ for $0 \leq i < n$, and therefore that s_n^Y is reachable. $\square$

5. Case description: customer support

In Australia, all telecommunications service providers that supply telecommunications products to Australian consumers must adhere to the Telecommunications Consumer Protections (TCP) code of conduct. The TCP code of conduct describes a large set of rules, including a number of obligations regarding the handling of complaints.

Fig. 7 illustrates a customer support process in the form of a marked net from one of the telecommunications providers that must adhere to the TCP code of conduct. Guards are included as conditions on the relevant activities (transitions). Silent, actionless, τ -transitions (marked in grey in the figure) are used to allow for the required control flow options.

The process starts when a complaint is received from a customer, after which the complaint is registered immediately. Depending on the urgency of the complaint, an enquiry is made immediately or within 48 h. In case the customer cannot be contacted, the complaint is closed and reported to the Telecommunications Industry Ombudsman (TIO) if relevant. In case there is contact, the complaint is confirmed and recorded, after which the credit management is suspended if the complaint involves a billing dispute. When the issue is easy to resolve, the customer is informed of a resolution. If the customer does not accept the offer, either a new offer can be made or the complaint can be escalated. If the complaint is not easily resolved, the customer is advised of the resulting process and timeframe, and the customer's details are confirmed. At this point, the issue is either administrative, technical, or both. Administrative issues are investigated and technical issues are assigned to Level 2 (L2) support. A resolution is formulated and any billing issues are resolved. Potential delays are recorded and discussed with the customer. If the customer chooses to accept the proposed resolution, the outcome is immediately recorded and the complaint is closed.

In order to ensure good service and fair outcomes for all customers, the process above must adhere to the TCP code of conduct. We highlight a number of those rules, before using them to evaluate our approach in Section 6.

1. Supplier must attempt to resolve a complaint on first contact by checking for immediate resolution.
2. Supplier must inform a customer, who is dissatisfied about resolution of a complaint, the options for escalation.

Fig. 7. The customer support process as marked net with guards (derived from [2]).

3. Supplier must advise customer of any delays of promised timeframes.
4. Supplier must ensure that relevant staff (i.e. administrative or technical staff) are made aware of the complaint.
5. Supplier must inform customer who is dissatisfied about resolution of a complaint of other possible resolutions.
6. Supplier must investigate the issue when not easily resolved.

We formalize the above rules as follows. Using the terms of the process, we first rewrite the rules to represent the form *when C holds within the process flow, then ϕ must also hold* and then use that condition C and rule ϕ to create the conditional CTL rule $\langle C, \phi \rangle$. For example, Rule 1 states “Supplier must attempt to resolve a

complaint”, which corresponds to either eventually performing t_9 or t_{19} where either an offer is being made or a resolution is formulated. The rule then continues by stating conditions in the form of “on first contact by checking for immediate resolutions”. These conditions directly correspond to the guard conditions of the process, in such a way that (i) there is contact and (ii) there exists an immediate resolution and therefore the complaint is considered easily resolved. In addition, these conditions imply that no resolution is required to be formulated because the resolution is immediate. Therefore, the rule does not apply to t_{19} which states a resolution is formulated. Given the above, we are able to rewrite Rule 1 to represent the form *when “there is contact and the issue can be easily resolved” holds within the process flow, then “supplier must*

eventually perform t_9 ” must also hold, or more formally, when “ $\text{contact} \wedge \text{easily resolved}$ ” holds within the process flow, then “ $\text{AF} t_9$ ” must also hold. From this, we then obtain the conditional CTL rule $\langle \text{contact} \wedge \text{easily resolved}, \text{AF} t_9 \rangle$. Applying the same method to the remaining rules, we can obtain the list of conditional CTL rules below:

1. $\langle \text{contact} \wedge \text{easily resolved}, \text{AF} t_9 \rangle$.
2. $\langle \text{contact} \wedge \neg \text{offer accepted}, \text{AF} t_{10} \rangle$.
3. $\langle \text{delay}, \text{AG}(t_{14} \Rightarrow \text{AF} t_{20}) \rangle$.
4. $\langle \text{contact} \wedge \neg \text{easily resolved} \wedge \text{tech fault}, \text{AF} t_{17} \rangle$ and $\langle \text{contact} \wedge \neg \text{easily resolved} \wedge \text{admin}, \text{AF} t_{18} \rangle$.
5. $\langle \neg \text{offer accepted} \wedge \text{review}, \text{AG}(t_{10} \Rightarrow \text{A}[t_{10} \cup \text{A}[\neg t_{10} \cup t_{10}]]) \rangle$.
6. $\langle \text{contact} \wedge \neg \text{easily resolved}, \text{AF}(t_{17} \vee t_{18}) \rangle$.

6. Evaluation

We implemented the proposed partial verification method in a Java package called *BPM Verification*.² The package takes as input a process model in PNML format³ extended with guards and a specification containing the CTL formulas to be evaluated. Subsequently, the conditional graph is generated and formally verified against its specifications using NuSMV2. NuSMV2 is a software tool for the formal verification of finite state systems using temporal logics.⁴

Using this toolchain, we conduct a series of evaluations of the proposed method. To qualitatively assess the applicability of the method, we first apply the proposed method on the case description presented in Section 5. Using the same case, we then evaluate the practical state space requirements of the proposed method compared to methods that consider the data perspective in process-centric business process models. Subsequently, we conduct a performance evaluation of the proposed method given highly complex synthetic business process models.

All tests were performed on a computer equipped with a quad core Intel® Core™ i7-7700HQ CPU @ 3.80 GHz, 16 GB RAM, running Ubuntu 18.10 and Java 1.8.0.181. Each test was executed five times, where the average time of three executions was recorded while removing the fastest and the slowest executions to eliminate load times.

6.1. Real-life example

To evaluate the applicability of the proposed approach, we apply it on the real-life case described in Section 5. Given this example, we apply the proposed method by verifying the compliance of the presented rules on the customer support process depicted in Fig. 7. The customer support process consists of 20 transitions representing activities and 8 τ -transitions that allow additional flow options. There are 21 guard conditions in total in the process, which complement or overlap each other. Given its initial marking and the guard conditions, there are 25 reachable markings.

To acquire a baseline for the sizes of the Kripke structures contained in the conditional graphs, we first obtain the full graph (i.e. the conditional graph with the condition $\{\text{true}\}$). The Kripke structure contained in this full graph has 37 states, 63 relations, and 28 atomic propositions and takes 5 ms to generate.

Next, we apply the conditions contained within each conditional CTL rule as conditions in the conditional graphs and evaluate its CTL formula on the Kripke structure obtained from that conditional graph. The results are shown in Table 1. The lower amount of states,

relations and atomic propositions clearly show the results of pruning unreachable states resulting from the conditions. The customer support process is compliant with all rules, except rule 6. This indicates that there is a scenario where t_{15} is not an OR-gate and both branches can theoretically be skipped (i.e. when `admin complaint` and `tech fault` are both `false`). This is clearly a design flaw that is easily identified when incorporating conditions in the rules.

Given the above, we conclude that process-centric methods that consider the full data perspective are not always required when evaluating *structural* properties of business processes with guards and are only required in cases where *behavioral* properties or variable values are considered. Instead, when evaluating structural properties of business process models, the proposed light-weight approach is suitable and provides the required information.

6.2. State space comparison

To evaluate the effective reduction of the state space complexity, we implemented a process-centric NuSMV2 model of the customer support process (Fig. 7) that also considers the data perspective of the business process (Appendix B). Based on the full graph, the NuSMV2 model is annotated with eleven boolean variables to extend the transition relations with the guard conditions included in the customer support process. Note that we use boolean variables instead of strings for those guard conditions that test string equivalency. We do this because booleans require the smallest meaningful data domains possible. Since the state space complexity of the data perspective scales exponentially to the size of the domain of the variables used, a model that uses booleans provides the *least complex solution*. As such, the comparison here is the minimum performance improvement of our reduction method (i.e. the performance improvement would generally be much larger when considering different variable types).

The resulting model is offered to NuSMV2, which reports a reachable state space of 75776 states. For the full graph, the reachable state space is reported to be 37 states. Indeed, these numbers are exactly what should be expected because the state space complexity of a process-centric business process model that considers the full data perspective is that of the state space complexity of the process-centric business process model times the state space complexity of the data perspective. That is, the expected state space complexity is that of 37 states multiplied by all possible combinations of the eleven boolean values, which is 75776 (i.e., $37 \times 2^{11} = 75776$). As a result, this method would require the seven CTL formulas to be evaluated over 75776 states. In contrast, the proposed method evaluated these seven CTL formulas over different partitions of a total number of 208 states (Table 1).

Given the worst case, with 2^{11} formulas with all different conditions which all have no impact on the reduction of the state space (i.e., there is no reduction of the 37 states due to a condition), the proposed method performs the same as process-centric methods that consider the full data perspective. More generically stated, when all possible combinations of values for all variables as conditions produce the full graph, the resulting state space is equal to that of the process-centric methods that consider the full data perspective. However, in this case all the produced graphs would be equal, and all formulas could be evaluated on a single full graph. Moreover, using the proposed method one is only ever interested in conditions that do have an impact on the reduction of the state space and thus result in Kripke structures with smaller state spaces. Furthermore, certain combinations of conditions considered within all possible combinations of values for all variables may result in Kripke structures that are equal. Such equal Kripke structures are produced because certain values for a single variable may determine a path at a guard that

² <https://github.com/rug-ds-lab/BPMVerification>.

³ <http://www.pnml.org/>.

⁴ <http://nusmv.fbk.eu>.

Table 1
Evaluation of the customer support process.

Specification	Conditional formula	S	R	AP	Result
1	(contact $\wedge$ easily resolved, AF t9)	15	20	14	true
2	($\neg$ offer accepted, AF t11)	33	52	24	true
3	(delay, AG(t14 $\Rightarrow$ AF t20))	36	60	27	true
4a	($\neg$ easily resolved $\wedge$ tech fault, AF t17)	30	46	23	true
4b	($\neg$ easily resolved $\wedge$ admin, AF t18)	28	41	23	true
5	($\neg$ offer accepted $\wedge$ review, AG(t10 $\Rightarrow$ A[t10 $\cup$ A[$\neg$ t10 $\cup$ t10]]))	33	54	24	true
6	(contact $\wedge$ $\neg$ easily resolved, AF(t17 $\vee$ t18))	33	53	24	false
Total		208	326	N/A	

only reaches guards that consider a subset of the remaining variables.

For example, consider the condition [contact = false $\wedge$ admin complaint = true] in Fig. 7. This combination of conditions produces a graph equal to that of the condition [contact = false] since no guard featuring a value of complaint is reachable when contact = false. Keeping this in mind, in practical settings the proposed method will always produce a total state space smaller than that of process-centric methods that consider the full data perspective when evaluating the structural properties of business processes with guards.

6.3. Performance evaluation

To test the scalability of our approach with increasingly complex process models, we adopt a collection of synthetic Petri nets of increasing size and different guards per path, and assess the approach with respect to performance and state space. The collection of synthetic models contains 4 different types: sequential models (SEQ), models with exclusive branches (XOR), models including concurrent branches (AND), and models with inclusive branches (OR). Each of these models have a number of n branches with a number of m tasks (i.e. transitions), with increasing n and m as measures to add complexity. Fig. 8 provides an overview of the basic constructs used to create the synthetic set of models, with the exception of SEQ as this is trivial.

The synthetic set of models includes two SEQ-models, one with length 5 and one with length 10. The guard expressions of the SEQ-models are of the form [$x=0$], and are on t_2 and t_5 for the models respectively. Subsequently, the condition for the CTL formulas is set to [$x=1$] (thereby contradicting the guards), such that the model contains guard in the middle of the sequence that causes a possible blocking run (and, hence, a reduced model can be obtained).

The expressions on the guards of each branch of an XOR-construct contain the same variable, but the expressions are mutually exclusive such that only 1 branch can be enabled at a time. As such, each guard expression is of the form [$x=i$], where i indicates the branch number, with branches ranging from 0 to $n-1$.

Finally, the guard expressions used in the AND- and OR-constructs are partially overlapping, using the same variable for every branch. Each guard expression is of the form [$x>i$], where i again indicates the branch number. Consequently, conditions for the CTL formulas determine whether the entire AND/OR-construct is taken into account, or a subset of branches. Every branch of the OR-construct has a complementing branch comprising a single τ -transition whose guard is the negation of the guard on the complementing branch to allow for multiple branches to be enabled while avoiding blocking runs due to the OR-join.⁵ For example, a

Table 2
Conversion algorithm results of models with n branches of m activities – full model.

#	Model Type	n	m	Kripke with [C=null]			Performance Full Kripke
				S	R	AP	
1	SEQ	1	5	6	6	5	0 ms
2	SEQ	1	10	11	11	10	0 ms
3	XOR	2	5	12	13	11	0 ms
4	XOR	3	5	17	19	16	0 ms
5	XOR	4	5	22	25	21	2 ms
6	XOR	4	10	43	46	42	2 ms
7	XOR	5	10	53	57	52	2 ms
8	AND	2	5	39	65	12	2 ms
9	AND	3	5	235	594	17	5 ms
10	AND	4	5	1515	5309	22	43 ms
11	AND	4	10	15370	56699	42	614 ms
12	AND	5	10	174418	822845	52	16990 ms
13	OR	2	5	52	93	14	1 ms
14	OR	3	5	361	973	20	11 ms
15	OR	4	5	2614	9951	26	346 ms
16	OR	4	10	21339	81101	46	1580 ms
17	OR	5	10	60926	1284039	57	55409 ms

guard expression [$x>1$] results in a guard expression [$x\leq 1$] on the corresponding τ -transition. For each construct, the branch count n ranges from 2 to 5, with branch length m ranging from 5 to 10.

The complexity of the verification is not in the size of the model in terms of number of activities in the model, but the total amount of possible executions that need to be verified $\times$ the length of each execution. The total amount of possible executions normally explodes due to heavy concurrency. Therefore, the most complicated model contains an OR-construct with 5×10 transitions. Even when we just assume that all 5 branches are activated, the total amount of possible executions for that model is $(5 \times 10)!/10!^5 = 4.83 \times 10^{31}$, which is well beyond the complexity typically encountered in real-life models.

The conditions paired with the CTL formulas cover various ranges for each experiment such that a subset of the branches can be taken into account. For each construct, we evaluate the conditional CTL formula $\langle C, \text{AF}t_{n-1,m} \rangle$ with condition C being [$x=n-1$] and $t_{n-1,m}$ being the last transition on branch number $n-1$. As a result, for XOR-constructs the CTL formula will be evaluated on the last branch, and for AND- and OR-constructs the CTL formula will be evaluated on the first $n-1$ branches. Given that the transition $t_{n-1,m}$ is on the last branch, the CTL formula will evaluate FALSE in 50% of the conditional test cases. Given the full graphs, the CTL formula will only evaluate FALSE for XOR-constructs.

The results of the experiment are shown in Tables 2 and 3. Although the transition graph allows further reduction through model equivalence with respect to stuttering [2,1], the table shows data prior the creation of such a reduced graph to highlight the differences between the generation of a full graph (i.e. ignoring conditions by assuming [true], Table 2) and the generation of the conditional graphs (Table 3).

⁵ Such a τ -transition does not exist for the AND-constructs, creating blocking runs. This is bad design and can be detected by the proposed approach when using a contradicting condition.

Fig. 8. Examples of the synthetic models used.

Table 3

Conversion algorithm results of models with n branches of m activities – with conditions.

#	Model Type	n	m	Reduced Kripke			Performance Partial Kripke
				C	S	R	
1	SEQ	1	5	$[x = 1]$	3	3	2 0 ms
2	SEQ	1	10	$[x = 1]$	6	6	5 0 ms
3	XOR	2	5	$[x = 1]$	7	7	6 0 ms
4	XOR	3	5	$[x = 2]$	7	7	6 0 ms
5	XOR	4	5	$[x = 3]$	7	7	6 1 ms
6	XOR	4	10	$[x = 3]$	13	13	12 1 ms
7	XOR	5	10	$[x = 4]$	13	13	12 1 ms
8	AND	2	5	$[x = 1]$	7	7	6 1 ms
9	AND	3	5	$[x = 2]$	38	64	11 1 ms
10	AND	4	5	$[x = 3]$	235	593	16 6 ms
11	AND	4	10	$[x = 3]$	1365	3728	31 66 ms
12	AND	5	10	$[x = 4]$	15369	56698	41 690 ms
13	OR	2	5	$[x = 1]$	16	23	8 1 ms
14	OR	3	5	$[x = 2]$	100	233	13 3 ms
15	OR	4	5	$[x = 3]$	688	2464	18 77 ms
16	OR	4	10	$[x = 3]$	3458	12384	33 115 ms
17	OR	5	10	$[x = 4]$	41474	205076	43 3392 ms

All experiments display a clear reduction of the state space complexity when comparing the full and partial Kripke structures, with most notable differences in the more complex models. This effect is explained by the interleaving of the parallel enabled transitions on concurrently executing branches. With fewer branches concurrently enabled due to the condition contradicting their guard expressions, the complexity from interleaving is reduced significantly. As a result, the impact of a condition on an interleaving of concurrently executing branches is directly related to the complexity of the interleaving itself.

In addition, the experiments show a decrease in conversion times with either equal or less time taken for graph generation. Although the less complex models show equal conversion times due to load times (despite attempts to reduce these by removing the slowest times), the more complex models display clear reductions of conversion times related to the reduced state space complexity of the partial Kripke structures. This is particularly noticeable for large concurrent models, like AND 5×10 and OR 5×10 , where by applying partial models the conversion time is reduced from 17 s to 0.7 s and from 55 s to 3.4 s respectively. As such, the effect of apply-

ing conditions with partial Kripke structures becomes significant for increasingly complex models.

7. Conclusion

In this paper, we presented a novel approach towards preventative compliance verification of business processes. The approach goes beyond existing approaches because of its ability to abstract and evaluate over conditional branches with non-propositional guards, without including the full data perspective. Most existing approaches completely lack support for verification over non-propositional guards or require the full data perspective which can quickly cause state space explosion problems. Instead, we have proven that by abstracting the guard conditions from both the graph and specification, the presented approach reduces state space complexity in the relevant cases where guard conditions affect the possible execution paths of the business process model.

The evaluation on a real life example from the Australian telecommunications sector clearly displays the value and applicability of the proposed approach. By abstracting guard conditions from the model and specifications through a condition and generating conditional graphs, a far larger spectrum of valuable insights can be obtained than without considering conditions. As such, the presented approach can be used by organizations to check if their business process models are compliant with established rules and regulations on a much more fine-grained level when compared to compliance checking without guards and conditions. The results show that the data perspective is not required when evaluating *structural* properties (i.e. control-flow) of business process models with guards and is only required in cases where *behavioral* properties or variable values are considered. As a result, when evaluating structural properties of business process models, the proposed light-weight approach is suitable and provides the required information. When evaluating data values or their evolution, the proposed approach is not suitable and the full data perspective is required. In addition, the proposed approach is not suitable when behavioral properties are being considered that cannot be inferred from the structural properties of the business process model.

Evaluation of the state space complexity demonstrates the value of the proposed approach. When evaluating structural properties of business process models, the proposed approach will always produce a total state space smaller than that of process-centric methods that consider the full data perspective in practical settings. Results show that for the real-life example from the Australian telecommunications sector the considered state space given seven rules is reduced from 75776 states using the full data perspective to different partitions of a total of 208 states using conditional graphs.

The performance evaluation illustrates the scalability of the approach with respect to its conversion from a Petri net to a conditional graph. Results show clear reductions in both the conversion times as well as the reduction in state space complexity when introducing conditional expressions, with most notable effects for the most complex models. As a matter of fact, when considering concurrently executing branches, the effect of conditional expressions over the guard conditions increases with the complexity of the model itself. As a result, the effect becomes increasingly significant with increasingly complex models.

For future work it would be interesting to include other real-life cases as well with more complex conditions, particularly when compared to existing approaches with a full data perspective. Finally, the implementation of the presented approach does not incorporate a visual user interface with graphical feedback yet. As future work, we plan to implement and support an increasing range of preventative compliance verification options with intuitive, human readable and visual design and feedback.

Appendix A. CTL semantics

The semantics of CTL are defined on a Kripke structure K using the minimal set of CTL operators $\{\neg, \vee, EX, EG, EU\}$.

Definition 8 (Semantics of CTL). $K, s_i \models \phi$ means that the formula ϕ holds at state s_i of the model K . When the model K is understood, $s_i \models \phi$ is written instead. The relation $\models$ is defined inductively as follows:

$s_i \models T$	iff	$s_i \not\models \perp$
$s_i \models p$	iff	$p \in L(s_i)$
$s_i \models \neg \phi$	iff	$s_i \not\models \phi$
$s_i \models \phi \vee \phi'$	iff	$s_i \models \phi$ or $s_i \models \phi'$
$s_i \models EX\phi$	iff	$\exists (s_j, s_{i+1}) \in R \mid s_{i+1} \models \phi$
$s_i \models EG\phi$	iff	$\exists \pi = s_i, s_{i+1}, s_{i+2}, \dots, \forall n : (n \geq 0 \wedge s_{i+n} \models \phi)$
$s_i \models E[\phi U \phi']$	iff	$\exists \pi = s_i, s_{i+1}, s_{i+2}, \dots, \exists m : (m \geq 0 \wedge s_{i+m} \models \phi' \wedge \forall n : (0 \leq n < m : s_{i+n} \models \phi))$

Appendix B. Data-driven model

```

MODULE main
VAR
urgent: boolean;
contact: boolean;
TIO: boolean;
billing: boolean;
easily: boolean;
accepted: boolean;
review: boolean;
tech: boolean;
CSG: boolean;
admin: boolean;
delay: boolean;
state: {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
S13, S16, S17, S18, S19, S20, S21, S22, S23, S27, S28,
S30, S34, S36, S42, S46, S50, S54, S60, S66, S72, S75,
S76, S78, S80};

```

```

DEFINE
t1:= (state = S0);
t2:= (state = S80);
t3:= (state = S1);
t4:= (state = S75);
t5:= (state = S76);
t6:= (state = S2);
t7:= (state = S3);
t8:= (state = S4);
t9:= (state = S5);
t10:= (state = S6);
t11:= (state = S7);
t12:= (state = S8);
t13:= (state = S9);
t14:= (state = S16);
t15:= (state = S17);
t16:= (state = S42) | (state = S54) | (state =
S21);
t17:= (state = S50) | (state = S46) | (state =
S23);
t18:= (state = S18) | (state = S42) | (state =
S66) | (state = S50) | (state = S19);
t19:= (state = S20);
t20:= (state = S22);
s1:= (state = S78);
s2:= (state = S72);
s3:= (state = S13);
s4:= (state = S11);
s5:= (state = S66) | (state = S60) | (state =
S27);
s6:= (state = S36) | (state = S54) | (state = S60)
| (state = S46) | (state = S34);
s7:= (state = S18) | (state = S36) | (state =
S30);
s8:= (state = S28);
ASSIGN
init(state) := {S0};
next(state) :=
case
state = S0 & urgent = TRUE: {S80};
state = S0 & urgent = FALSE: {S1};
state = S1 & contact = FALSE: {S75};
state = S1 & contact = TRUE: {S2};
state = S2: {S3};
state = S3 & billing = TRUE: {S4};
state = S3 & billing = FALSE: {S72};
state = S4 & easily = FALSE: {S16};
state = S4 & easily = TRUE: {S5};
state = S5 & accepted = TRUE: {S13};
state = S5 & accepted = FALSE: {S6};
state = S6 & review = FALSE: {S7};
state = S6 & review = TRUE: {S11};
state = S7: {S8};
state = S8: {S9};
state = S9: {S10};
state = S10: {S10};
state = S11 & accepted = TRUE: {S13};
state = S11 & accepted = FALSE: {S6};
state = S13: {S8};
state = S16: {S17};
state = S17 & admin = FALSE
& tech = TRUE
& CSG = TRUE: {S54};
state = S17 & admin = TRUE
& tech = TRUE
& CSG = FALSE: {S66};

```



```

state = S17 & admin = FALSE
& tech = TRUE
& CSG = FALSE: {S60};
state = S17 & admin = TRUE
& tech = FALSE: {S18};
state = S17 & admin = TRUE
& tech = TRUE
& CSG = TRUE: {S42};
state = S17 & admin = FALSE
& tech = FALSE: {S36};
state = S18 & tech = FALSE: {S30};
state = S18 & tech = TRUE
& CSG = TRUE: {S21};
state = S18 & tech = TRUE
& CSG = FALSE: {S27};
state = S18 & admin = FALSE: {S34};
state = S18 & admin = TRUE: {S19};
state = S19: {S20};
state = S20 & delay = FALSE: {S28};
state = S20 & delay = TRUE: {S22};
state = S21: {S23};
state = S22 & accepted = TRUE: {S13};
state = S22 & accepted = FALSE: {S6};
state = S23: {S20};
state = S27: {S23};
state = S28 & accepted = TRUE: {S13};
state = S28 & accepted = FALSE: {S6};
state = S30: {S20};
state = S34: {S20};
state = S36 & tech = FALSE: {S30};
state = S36 & tech = TRUE
& CSG = TRUE: {S21};
state = S36 & admin = FALSE: {S34};
state = S36 & tech = TRUE
& CSG = FALSE: {S27};
state = S36 & admin = TRUE: {S19};
state = S42 & admin = FALSE: {S46};
state = S42 & tech = FALSE: {S30};
state = S42 & admin = TRUE: {S50};
state = S42 & tech = TRUE
& CSG = TRUE: {S21};
state = S42 & tech = TRUE
& CSG = FALSE: {S27};
state = S46 & admin = FALSE: {S34, S23};
state = S46 & admin = TRUE: {S19, S23};
state = S50 & admin = FALSE: {S34, S23};
state = S50 & admin = TRUE: {S19, S23};
state = S54 & admin = FALSE: {S46};
state = S54 & admin = TRUE: {S50};
state = S54 & tech = FALSE: {S30};
state = S54 & tech = TRUE
& CSG = TRUE: {S21};
state = S54 & tech = TRUE
& CSG = FALSE: {S27};
state = S60 & admin = FALSE: {S46};
state = S60 & admin = TRUE: {S50};
state = S60 & tech = FALSE: {S30};
state = S60 & tech = TRUE
& CSG = TRUE: {S21};
state = S60 & tech = TRUE
& CSG = FALSE: {S27};
state = S66 & admin = FALSE: {S46};
state = S66 & tech = FALSE: {S30};
state = S66 & admin = TRUE: {S50};
state = S66 & tech = TRUE
& CSG = TRUE: {S21};

```

```

state = S66 & tech = TRUE
& CSG = FALSE: {S27};
state = S72 & easily = FALSE: {S16};
state = S72 & easily = TRUE: {S5};
state = S75 & TIO = TRUE: {S76};
state = S75 & TIO = FALSE: {S78};
state = S76: {S10};
state = S78: {S10};
state = S80 & contact = FALSE: {S75};
state = S80 & contact = TRUE: {S2};
esac;

```

References

- Groefsema, H., van Beest, N.R.T.P., 2015. Design-time compliance of service compositions in dynamic service environments. *International Conference on Service Oriented Computing & Applications*, 108–115.
- Groefsema, H., van Beest, N.R.T.P., Aiello, M., 2018. A formal model for compliance verification of service compositions. *IEEE Trans. Serv. Comput.* 11, 466–479.
- Groefsema, H., van Beest, N.R.T.P., Armas-Cervantes, A., 2017. Automated compliance verification of business processes in Apromore. *Proceedings of the BPM Demo Track*.
- van Beest, N.R.T.P., Groefsema, H., Garcia-Banuelos, L., Aiello, M., 2018. Variability in business processes: automatically obtaining a generic specification. *Inf. Syst.*, To appear.
- Groefsema, H., Bucur, D., 2013. A survey of formal business process verification: from soundness to variability. *International Symposium on Business Modeling and Software Design*, 198–203.
- Lam, V.S., 2010. Formal analysis of BPMN models: a NUSMV-based approach. *Int. J. Softw. Eng. Knowl. Eng.* 20, 987–1023.
- Kherbouche, O.M., Ahmad, A., Basson, H., 2013. Using model checking to control the structural errors in BPMN models. *IEEE 7th International Conference on Research Challenges in Information Science (RCIS)*, 1–12.
- Dechsupa, C., Vatanawood, W., Thongtak, A., 2018. Transformation of the BPMN design model into a colored petri net using the partitioning approach. *IEEE Access* 6, 38421–38436.
- Governatori, G., Milosevic, Z., Sadiq, S., 2006. Compliance checking between business processes and business contracts. *Enterprise Distributed Object Computing Conference*, 221–232.
- Goedertier, S., Vanthienen, J., 2006. Designing compliant business processes with obligations and permissions. *Proc. Int. Conf. on Business Process Management Workshops, BPM*, 5–14.
- Ghose, A., Koliadis, G., 2007. Auditing business process compliance. *Int. Conf. on Service-Oriented Computing*, 169–180.
- Bianculli, D., Ghezzi, C., Spoletini, P., 2007. A model checking approach to verify BPEL4WS workflows. *Int. Conf. on Service-Oriented Computing and Applications*, 13–20.
- Weber, B., Reichert, M., Rinderle-Ma, S., 2008. Change patterns and change support features – enhancing flexibility in process-aware information systems. *Data Knowl. Eng.* 66, 438–466.
- Favre, C., Völzer, H., 2010. Symbolic execution of acyclic workflow graphs. *Int. Conf. on Business Process Management*, 260–275.
- Montali, M., Torroni, P., Chesani, F., Mello, P., Alberti, M., Lamma, E., 2010. Abductive logic programming as an effective technology for the static verification of declarative business processes. *Fund. Inf.* 102, 325–361.
- Kheldoun, A., Barkaoui, K., Ioualalen, M., 2015. Specification and verification of complex business processes – a high-level Petri net-based approach. *Int. Conf. on Business Process Management*, 55–71.
- Awad, A., Weidlich, M., Weske, M., 2009. Specification, verification and explanation of violation for data aware compliance rules. *Service-Oriented Computing*, 500–515.
- Knuplesch, D., Ly, L.T., Rinderle-Ma, S., Pfeifer, H., Dadam, P., 2010. On enabling data-aware compliance checking of business process models. *International Conference on Conceptual Modeling*, 332–346.
- Deutsch, A., Hull, R., Patrizi, F., Vianu, V., 2009. Automatic verification of data-centric business processes. *International Conference on Database Theory*, 252–267.
- Deutsch, A., Hull, R., Li, Y., Vianu, V., 2018. Automatic verification of database-centric systems. *ACM SIGLOG News* 5, 37–56.
- Szpyrka, M., Nalepa, G.J., Kluz, K., 2017. From process models to concurrent systems in alvis language. *Informatica* 28, 525–545.
- Monakova, G., Kopp, O., Leymann, F., Moser, S., Schödl, K., 2009. Verifying business rules using an SMT solver for BPEL processes. In: *Business Process, Services-Computing and Intelligent Service Management*.
- Rachdi, A., En-Nouaary, A., Dahchour, M., 2016. Analysis of common business rules in BPMN process models using business rule language. *2016 11th International Conference on Intelligent Systems: Theories and Applications (SITA)*, 1–6.
- Petri, C., 1966. *Communication with Automata*. Universit”at Hamburg, Ph.D. thesis.

- Reisig, W., Rozenberg, G., 1998. Lectures on Petri Nets I: Basic Models: Advances in Petri Nets. Springer Science & Business Media.
- Clarke, E., Grumberg, O., Peled, D., 1999. Model Checking. The MIT Press, Cambridge, Massachusetts and London, UK.
- Emerson, E., Halpern, J., 1982. Decision procedures and expressiveness in the temporal logic of branching time. Proc. of the 14th Annual ACM Symposium on Theory of Computing, 169–180.
- Vardi, M.Y., 2001. Branching vs. linear time: final showdown. Tools and Algorithms for the Construction and Analysis of Systems, 1–22.
- Trčka, N., Van der Aalst, W.M.P., Sidorova, N., 2009. Data-flow anti-patterns: discovering data-flow errors in workflows. International Conference on Advanced Information Systems Engineering, 425–439.