University of Groningen

Variability in business processes

van Beest, Nick; Groefsema, Heerko; Garcia-Banuelos , Luciano; Aiello, Marco

Published in:
Information Systems

DOI:
[10.1016/j.is.2018.09.005](https://doi.org/10.1016/j.is.2018.09.005)

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:
2019

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

Citation for published version (APA):

van Beest, N., Groefsema, H., Garcia-Banuelos , L., & Aiello, M. (2019). Variability in business processes: Automatically obtaining a generic specification. *Information Systems*, 80, 36-55.
<https://doi.org/10.1016/j.is.2018.09.005>

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.

Variability in business processes: Automatically obtaining a generic specification

Nick van Beest^{a,*}, Heerko Groefsema^{b,*}, Luciano García-Bañuelos^c, Marco Aiello^d

^a Data61, CSIRO, Australia

^b Bernoulli Institute, University of Groningen, The Netherlands

^c University of Tartu, Tartu, Estonia

^d Institute of Architecture of Application Systems, University of Stuttgart, Germany

HIGHLIGHTS

- Automatically derive declarative variability rules from business process variants.
- Behavioral relations from different process variants are combined and integrated.
- The rules allow any variant in the design space defined by the input variants.
- The rules can be used to automatically verify the compliance of a process variant.

ARTICLE INFO

Article history:

Received 24 January 2018

Received in revised form 17 August 2018

Accepted 26 September 2018

Available online 3 October 2018

Keywords:

Business process model

Declarative variability modeling

Event structure

Temporal logic

ABSTRACT

The existence of different process variants is inevitable in many modern organizations. However, variability in business process support has proven to be a challenge as it requires a flexible business process specification that supports the required process variants, while at the same time being compliant with policies and regulations. Declarative approaches could support variability, by providing rules constraining process behavior and thereby allowing different variants. However, manual specification of these rules is complicated and error-prone. As such, tools are required to ensure that duplication and overlap of rules is avoided as much as possible, while retaining maintainability. In this paper, we present an approach to represent different process variants in a single compound prime event structure, and provide a method to subsequently derive variability rules from this compound prime event structure. The approach is evaluated by conducting an exploratory evaluation on different sets of real-life business process variants, including a real-life case from the Dutch eGovernment, to demonstrate the effectiveness and applicability of the approach.

© 2018 Elsevier Ltd. All rights reserved.

1. Introduction

The existence of different process variants is inevitable in many modern organizations and in most cases even a deliberate choice. Process variants emerge naturally due to mergers, handling of different products, customer-tailored services, differences in market segments or in legislation across different countries. As such, tools are required to help business analysts to analyze different variants together and coherently describe each variant in such a way that duplication of specifications is avoided as much as possible, while retaining maintainability.

For example, there are 380 municipalities in the Netherlands, which all differ considerably with respect to demographics, economics, etc. As a result, each municipality is allowed to operate independently according to their local requirements. However, all municipalities have to provide the same services and execute the same laws. An example of such a law that is heavily subjected to local needs is the WMO (Wet Maatschappelijke Ondersteuning, Social Support Act, 2006). This law provides citizens with support ranging from wheelchairs, help at home, to home accessibility improvement. Consequently, each municipality requires its own variant of the business process supporting this law.

In general, there are two distinct approaches to support variability. The imperative approach tries to merge all variants and provide a so-called *configurable process model* that can be configured pre-runtime to suit the exact needs and, thereby, generate the specific variant. The declarative approach, on the other hand, provides rules in which the allowed process behavior is implicitly

* Corresponding authors.

E-mail addresses: nick.vanbeest@data61.csiro.au (N. van Beest), h.groefsema@rug.nl (H. Groefsema), luciano.garcia@ut.ee (L. García-Bañuelos), marco.aiello@iaas.uni-stuttgart.de (M. Aiello).

represented. In this case, anything that is not constrained by rules is allowed.

Imperative approaches quickly lead to very complex consolidated models, which are difficult to maintain. Declarative approaches, however, pose some design challenges as for complex processes the amount of rules can grow considerably and the specification of a coherent set of rules allowing each required variant (while being compliant to laws and regulations) is all but straightforward.

These rules can be, for example, (if D then either E or F should occur) or (E should always be followed by F). In Fig. 1, two excerpts¹ are shown of distinct variants of a municipal process.² In both variants, F occurs after D eventually (i.e. some activities may occur in between), but the order of activities in the second variant is different: E may be followed by F eventually, but in that case U is to be executed first. In variant 1, however, E is always followed by F and U does not exist.

Although a natural language representation of such rules (like shown here) is convenient with respect to readability, they are typically formulated in technical terms like temporal logic, in order to allow those rules to be handled by a process engine. In addition, the example here only shows excerpts of processes and variant 1 already requires 10 rules to describe all allowed behavior and 16 rules to include the behavior allowed by variant 2 as well. For complex business processes, the total amount of rules may easily approach 1000. As such, it is virtually impossible for a business analyst to compose and maintain these rules by hand. Consequently, an approach is required to automatically derive variability rules from all variants, such that the commonalities can be captured to provide a generic set of declarative rules supporting all process variants.

In this paper, we present an approach to automatically derive declarative variability rules describing a set of business process variants. As shown graphically in Fig. 2, we first obtain the behavior relations of each process variant. Subsequently, the common behavior relations for all process variants are identified. Finally, we automatically derive declarative variability rules from the set of common behavior relations. The declarative variability rules are expressed in temporal logic, and define the specification representing all supported business process variants.

As such, the contribution of this paper is twofold: (i) it shows how to represent different process variants by a set of common behavior relations, and (ii) it provides a method to automatically derive declarative variability rules from these behavior relations.

The contribution of this paper is complementary to the work presented in [1] (marked in gray in Fig. 2), where an efficient approach is developed to automatically check whether a certain business process is compliant to a set of rules. That is, the rules generated by the approach presented here are compatible with the rules used in the method from [1], such that any other process variant can be verified on the set of declarative variability rules to check whether this variant is supported. As such, the methodology presented in this paper allows to check the conformance of a new process variant to an existing family of business process variants.

The remainder of the paper is structured as follows. First, Section 2 discusses a case study from the Dutch local eGovernment, which is used as a running example throughout this paper. Section 3 describes the formal translation from a process model into a set of behavior relations, which are used in Section 4 to compose

a set of common behavior relations describing a set of business process variants. Subsequently, Section 5 introduces the formal procedure to obtain the declarative variability rules. The proposed approach is evaluated with respect to its complexity, performance and applicability to new process variants in Section 6 using the process variants introduced in Section 2 and an online data set of different business process variants. Finally, Section 7 discusses related work, before the paper is concluded in Section 8.

2. Case study: Local eGovernment

In the Netherlands, the WMO (Wet Maatschappelijke Ondersteuning, Social Support Act, 2006) is a law that, although defined nationally, is executed locally at the municipalities. As such, it is heavily subjected to local needs, resulting in different variants of the same process for each of the 380 municipalities that exist in the Netherlands.

Figs. 3–5 illustrate the main process flows of the WMO process observed at three distinct municipalities. The illustrated processes were obtained through interviews with different municipalities located in the Northern area of the Netherlands [2,3]. The interviewed municipalities differed with respect to size, population, income, and in being urban or rural.

Fig. 3 depicts the simplest variant of the three WMO processes. The process starts with an application procedure that determines if the request made by the citizen falls under the WMO law. If this is not the case, the citizen is advised by the municipality employee towards his next steps. When the request made by the citizen is eligible for the WMO law, the application is accepted, and a decision whether to approve the requested provision is made based upon the intake, possible requests of medical advice, and a possible home visit. The request is then either approved or rejected. In case of a positive decision, the requested provision is either arranged directly for the citizen, or a personal budget is assigned in case of personal care. In case of a negative decision, the citizen can appeal the decision. If the appeal is found to have merit, the decision is revised and the process is restarted.

Fig. 4 depicts a second variant of the WMO process. The main difference of this variant, when compared to the variant illustrated in Fig. 3, is the option to approve the requested provision in part. When this option is selected, further medical advice may be acquired. When the medical advice declares that the citizen qualifies for the requested provision in full, the decision to approve in part is revised. Another difference is included after the reverse decision option in case of an appeal. Instead of restarting the process, this variant only revisits the approve or reject options. Finally, an activity is included to provide the citizen with information in case a personal budget is assigned.

A third variant of the WMO process is illustrated in Fig. 5 and demonstrates a large amount of diversity. In this variant, a new branch is introduced for Directly Executable Provisions (DEP). In case of DEP, the requested provision is directly provided without taking the long decision making procedure. The information gathering process before making a decision also includes an option to do file research. In addition, each activity in this process can be performed an arbitrary number of times and in any order. All decisions in this variant are verified as well, and can be revised before the official approval or rejection. Similar to the second variant, an option to approve in part is included. In this case, however, the option is included after the rejection of the initially requested provision. Similar to the second variant, the citizen is also provided with information in case a personal budget is assigned. In addition, a rejection is explained after a failed appeal by a citizen. Finally, the activity of acquiring requirements of any assigned provision is handled by the supplier without involvement of the municipality.

When analyzing the three respective business processes illustrated in Figs. 3–5, we can clearly see the same generic process flow

¹ The full processes will be described in detail in Section 2.

² The processes are depicted in Petri net notation, containing *places* (circles representing conditions) and *transitions* (squares representing activities). Places may contain *tokens*, indicated by a black dot. A transition is enabled and can “fire” when all its input places have at least one token. When a transition fires, one token is removed from each input place and one token is put into each output place. A formal definition is provided in Section 3.

Fig. 1. Example of variants.

Fig. 2. Method overview.

Fig. 3. WMO provision request Petri net of Municipality A.

in all three versions of the WMO process, as we highlighted the unique features of each model in orange. Starting at the intake and application, the generic process flow continues through decision, approval, and finally the assignment of either the requested provision or a personal budget. Similar generic process flows can be seen at the alternate information and advice path and the reject/appeal path. Variations appear at the information gathering stage of the process, where the file research, home visit, and medical advice activities contribute to understanding the background of the citizens making the WMO provision request. Other variations appear through additional paths for approval in part, simple provision requests (i.e., DEP requests), and decision verification. Finally, the simplest variations appear through the inclusion of additional informative activities in existing paths.

3. Obtaining behavior relations from a process

There exist multiple mainstream notations used in practice to model business processes, which include (among others) BPMN, EPC, and Petri nets. In this paper, Petri nets are used as we are building on previous work, capitalizing the rich body of theory and tools that come with this formalism. Fortunately, the mapping of BPMN and other notations to Petri nets has been extensively studied elsewhere (e.g. [4–6]).

As stated in the introduction, we aim to derive declarative variability rules from a set of input process models. From the perspective of a declarative set of rules, however, the allowed control-flow is specified by the relations between activities with respect to their allowed or constrained behavior. That is, rules are used to specify what activities are required, what activity sequences are not allowed in one run (conflict), which activities can be executed

Fig. 4. WMO provision request Petri net of Municipality B.

in parallel, etc. As such, the transformation of a set of process models into a set of declarative rules requires so-called *behavior relations* to be obtained from each input model. These behavior relations are then combined into a set of common behavior relations that together represent the generic process model (as explained in Section 4).

The required behavior relations cannot always be derived directly from process models because they may mix parallelism and conflict over very complex topologies. Building up on early work [7], we take the input model represented as a Petri net and compute its corresponding set of runs, collectively represented as a tree-like structure called a *branching process*. However, runs may be infinite due to the presence of loops. Therefore, we compute a finite prefix of a branching process, referred to as the *complete prefix unfolding*, which is later expanded to represent one repetition of each loop in the model at the most.

3.1. Petri nets and prefix unfoldings

Petri nets are directed digraphs containing *places* (circles representing conditions) and *transitions* (squares representing activities). *Arcs* form directed edges between place and transition pairs. Places may contain *tokens*. A distribution of tokens over places is called a *marking*. To execute a Petri net, we say a transition is enabled and can “fire” when all its input places have at least one token. When a transition fires, one token is removed from each input place and one token is put into each output place. Throughout the paper, the transitions of a Petri net are labeled to explicitly represent the name of the activity on each transition. A *labeled Petri net* is defined formally as follows [8,9]:

Fig. 5. WMO provision request Petri net of Municipality C.

Definition 1 (Labeled Petri Net). A tuple (P, T, A, λ) is a *labeled Petri net*, where:

- P is a set of *places*
- T is a set of *transitions*, such that $P \cap T = \emptyset$
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs
- $\lambda : T \rightarrow \mathcal{L} \cup \{\tau\}$ is a labeling function, with $\mathcal{L}$ being a set of labels. A transition is said to be *silent* (representing internal, unobservable system behavior), if $\lambda(t) = \tau$.

The Petri net state, often referred to as the *net marking* $M : P \rightarrow \mathbb{N}_0$ is a function that associates a place $p \in P$ with a natural number (viz., place tokens). A *labeled marked net* $N = (P, T, A, \lambda, M_0)$ is a Petri net (P, T, A, λ) together with an *initial marking* M_0 . A Petri net is said to be *safe* iff places hold at most one token in every reachable marking.

Places and transitions are referred to as *nodes*. The *preset* of a node is denoted by $\bullet y = \{x \in P \cup T \mid (x, y) \in A\}$, and the *postset* of a node is denoted by $y \bullet = \{z \in P \cup T \mid (y, z) \in A\}$. A^+ and A^* denote the irreflexive and reflexive transitive closure of A , respectively. In the following, we assume that all the Petri nets are labeled, such that we omit the word “labeled”.

The semantics of a Petri net are defined in terms of the set of all possible markings of it, collected as a labeled transition

system which is referred to as the reachability graph. In this work, however, we rely on an alternative formulation of the semantics of a Petri net that uses another Petri net to explicitly represent all possible execution paths of the original net in a single tree-like structure, called a *branching process* [10,11].

Fig. 6(a) and (b) present a Petri net and an excerpt of its branching process side-by-side, which will be used later on as a running example. Intuitively, the branching process is a net that can be built to track the execution of a given Petri net (e.g. according to the well-known token game, where one fires transitions and moves tokens among places). Fig. 8 illustrates the stepwise construction of the branching process in Fig. 6(b).

Note that the branching process does have forking places but no merging places. Petri nets with such a structure are referred to as occurrence nets. Interestingly, the behavior relations concurrency, conflict and causality can be precisely defined over occurrence nets as follows:

Definition 2 (Occurrence Net, Behavior Relations). Let $N = (P, T, A, \lambda)$ be a Petri net. N is said to be an *occurrence net* iff every place p has at most one predecessor (i.e. $|\bullet p| \leq 1$) and has zero or more successors (i.e. $|p \bullet| \geq 0$). Moreover, let $x, y \in P \cup T$ be two nodes in N .

- The *identity* relation states that x is identical to itself, denoted $x = x$,
- x and y are *causal*, written $x \leq_N y$, iff $(x, y) \in A^+$,
- x and y are in *conflict*, denoted $x \#_N y$, iff $\exists t, t' \in T : t \neq t' \wedge \bullet t \cap \bullet t' \neq \emptyset \wedge (t, x), (t', y) \in A^*$,
- x and y are *concurrent*, denoted $x \parallel_N y$, iff neither $x \leq_N y$, $y \leq_N x$, $x \#_N y$, nor $x = y$.

Armed with the above, we can now formally present a method to construct the branching process of a given Petri net [11]:

Definition 3 (Branching Process). Let $N = (P, T, A, \lambda_N, M_0)$ be a labeled Petri net. The *branching process* $\beta = (B, E, G, \lambda_\beta, \rho)$ of N is defined by the inductive rules in Fig. 7, where:

- B is the set of conditions,
- E is the set of events,
- G denotes the flow relation of the branching process,
- $\rho : B \cup E \rightarrow P \cup T$ is a function that maps each node in β to a node in N , and
- λ_β denotes the labeling of events in the branching process and is computed by composing λ_N and ρ , i.e. $\lambda_\beta = \lambda_N \circ \rho$.

With abuse of notation, we will use $\rho(X)$ as a shorthand for $\{\rho(x) \mid x \in X\}$, with $X \subseteq B \cup E$. Moreover,

For sake of clarity, the elements of B and E in a branching process β are respectively called *conditions* and *events*. G in turn denotes the flow relation of the branching process. $\text{Min}(\beta)$ denotes the set of minimal elements of $B \cup E$ with respect to the transitive closure of G . As such, $\text{Min}(\beta)$ corresponds to the set of places in the initial marking of N , i.e., $\rho(\text{Min}(\beta)) = M_0$.

Fig. 8 illustrates the construction of the branching process of the running example in a stepwise manner. According to rule (1), a new condition is added to the branching process for each place in the initial marking of the Petri net. In the running example, there is only one place in the initial marking. Henceforth, only one condition is added to the branching process as illustrated in Fig. 8(a). According to rule (2), we select a set of (concurrent) conditions and add a new event into the branching process. Additionally, according to rule (3) we would add a new condition for every output place to the transition being processed. Applying rules (2) and (3) will result in the branching process shown in Fig. 8(b). The event added Fig. 8(b) is referred to as event e_0 , and rule (2) also state that

$\rho(e_0) = A$. Moreover, by composing ρ and λ_N we would carry the labeling of nodes in β , i.e. $\lambda_\beta(e_0) = \lambda_N(A) = A$. We would proceed by selecting the conditions that hold a token in Fig. 8(b), which we have added in the figure to help in referencing the conditions under consideration. Note that the conditions are concurrent (according to Definition 2) and that tokens in the corresponding places over the original net would enable transitions B , C and D . The three possible events that can be added to the branching process are referred to as possible extensions. In the figure, we illustrate how each one of the possible extensions are added to the branching process one after the other, applying iteratively rules (2) and (3) ad infinitum.

Indeed, the branching process of a Petri net with cycles (like our example) is infinite. For safe nets, however, a *prefix* of a branching process fully encodes the behavior of the original net, as shown in [12]. Such a prefix is referred to as the *complete prefix unfolding* of a net. We will use β_m to denote a maximal, possibly infinite, branching process and β_f to denote a complete prefix unfolding of β_m . The work reported in [11], which in turn refines the one reported in [12], defines criteria to identify a set of events in the prefix of an unfolding where the process can be stopped with the assurance that the entire behavior of the original Petri net is captured thereof. We refer the reader to the original paper [11] to gain a full understanding on how to compute the complete prefix unfolding β_f .

The set of events that are identified as the ones where the unfolding can be stopped are referred to as *cutoff events*. In the branching process presented in Fig. 6(b), e_4 , e_6 and e_7 are cutoff events. If we consider e_4 , we can see that the unfolding can be stopped after e_4 because the output conditions of e_4 can be paired with the output conditions of e_0 (the colors used with the conditions hint their correspondence). Intuitively, we can say that the marking produced after e_4 is already represented in the prefix, such that continuing unfolding the net would add information that is redundant. Note that e_0 is referred to as the *corresponding event* of the cutoff events e_4 and e_6 . Similarly, the cutoff event e_7 has e_5 as its corresponding event. In the following, we will refer to pairs of cutoff/corresponding events as *cc-pairs*.

In [13], Nielsen et al. showed that behavior relations derived from occurrence nets, collected in the so-called *prime event structure*, represent faithfully the behavior encoded by the original net. Interestingly, a method to derive an occurrence net from a prime event structure is also in [13], but that is out of the scope of this work. The events in the branching process are identified as events in the event structure and, by explicitly representing the behavior relations in the event structure, we can just abstract away the conditions. Such an intuition is formally captured in the following definition:

Definition 4 (Prime Event Structure). Let $\beta = (B_\beta, E_\beta, G_\beta, \lambda_\beta, \rho_\beta)$ be the branching process of the net system N . Moreover, let $\leq_\beta$, $\#_\beta$ and $\parallel_\beta$ be the behavior relations of β . The *prime event structure* of β (and hence of N) is the tuple $\mathcal{E}(N) = (E, \leq, \#, \parallel, \lambda_\mathcal{E})$ where:

- E is the set of events of $\mathcal{E}(N)$ and corresponds to E_β , i.e. $E = E_\beta$,
- $\leq$ is the causality relation restricted over E , i.e. $\leq = \leq_\beta \cap E_\beta \times E_\beta$,
- $\#$ is the conflict relation restricted over E , i.e. $\# = \#_\beta \cap E_\beta \times E_\beta$,
- $\parallel$ is the concurrency relation restricted over E , i.e. $\parallel = \parallel_\beta \cap E_\beta \times E_\beta$, and
- $\lambda_\mathcal{E}$ is the labeling function restricted over E , i.e. $\lambda_\mathcal{E} = \lambda_\beta|_{E_\beta}$.

To represent the behavior specified by a process model, in this work we use the prime event structure derived from the complete prefix unfolding of the corresponding Petri net, which is referred to as the *PES prefix unfolding* (PES prefix for short) of

Fig. 6. Sample Petri net and its corresponding branching process.

$$\begin{array}{ll}
 \frac{p \in M_0}{b = \langle \emptyset, p \rangle \in B \quad \rho(b) = p} & (1) \text{ Min. elements} \\
 \frac{t \in T \quad B' \subseteq B \quad B' \times B' \subseteq \parallel_\beta \quad \rho(B') = \bullet t}{e = \langle B', t \rangle \in E \quad \rho(e) = t} & (2) \text{ Adding events} \\
 \frac{e = \langle B', t \rangle \in E \quad t \bullet = \{p_1, \dots, p_n\}}{b_i = \langle t', p_i \rangle \in B \quad \rho(b_i) = p_i} & (3) \text{ Adding conditions}
 \end{array}$$

Fig. 7. Inductive construction of a branching process.

Fig. 8. Stepwise construction of the branching process in Fig. 6.

Fig. 9. Excerpt of the PES prefix unfolding of Fig. 6.

a model. In [14], it is shown that silent events can be removed in a behavior-preserving manner, under the well-known notion of visible-pomset equivalence.³ The PES prefix unfolding of Fig. 6 is shown graphically in Fig. 9.

In the graphical representation of an event structure, it is common to represent only the direct causal relations with straight

directed arcs, and not the full transitive causality relations to avoid cluttering. For instance, we can see a directed arc connecting $e_0 : A$ and $e_1 : B$ implying $e_0 \leq e_1$, and also one connecting $e_1 : B$ and $e_4 : \tau$ implying $e_1 \leq e_4$. Moreover, since causality is transitive we know that $e_0 \leq e_4$, although this fact is not explicitly represented. The conflict relation is represented with dotted undirected edges. For instance, we can infer from the graphical representation that $e_1 \# e_3$ by the presence of the dotted edge. The conflict relation is said to be “hereditary”, meaning that it is propagated through out the transitive causality. For instance, we can infer that e_3 and e_5 are in conflict because $e_2 \# e_3 \wedge e_2 \leq e_5$ holds. Every pair of events with out any direct or transitive causal or conflict relation is considered as concurrent. This is the case of events e_1 and e_2 in the example above. Finally, the PES prefix also represents the cc-pairs with red dashed arcs. The origin of a cc-pair arc is the cutoff event, and the head of the arc points to the corresponding event. The definition below extends Definition 4 to carry over the information about cc-pairs on a complete prefix unfolding.

Definition 5 (PES Prefix Unfolding). Let $\beta_f = (B_f, E_f, G_f, \lambda_{\beta_f}, \rho)$ be the complete prefix unfolding of the net system N . Let $\bar{E}_f \subseteq E_f$ be

³ This holds if every sink event in the event structure is a visible event, which can be ensured by adding a dummy labeled final event to the Petri net from which the event structure is generated.

the set of events of β_f such that $e \in \bar{E}_f$ iff e is labeled (i.e. $\lambda_\beta(e) \neq \tau$), or e is a cutoff or corresponding event. The PES prefix unfolding of N , denoted $\bar{E}(N)$, is defined as:

$$\bar{E}(N) \triangleq (\bar{E}_f, \leq_\beta \cap \bar{E}_f^{-2}, \#_\beta \cap \bar{E}_f^{-2}, \lambda_\beta|_{\bar{E}_f})$$

3.2. Identifying possible executions and elementary cycles

Since the unfolding of a branching process is stopped when the prefix captures all existing states, it might not explicitly represent all the behavior relations held between pairs of transitions in the original Petri net. For instance, the occurrence of task B may be observed causally preceding the occurrence of task D in a run where the loop body is repeated once. Instead, in the PES prefix we can only perceive the fact that if the loop body is executed only once, the run would include one occurrence of task B or one occurrence of task D , but not both.

To overcome this limitation, we need to expand the PES prefix to a structure where all the looping behavior is explicitly represented. In the case of our running example, this would mean to build the PES prefix shown in Fig. 10. There, we can clearly see that one occurrence of B (e_1) may causally precede one of D (e_{10}) if the loop iterates once.

Before discussing how the PES prefix is extended, we need to introduce some other concepts. A state in an execution over a branching process or a PES can also be described in terms of sets of events that can occur together, and it is referred to as a *configuration*. For instance, the state of the execution represented by the branching process shown in Fig. 8(b) and (c) can be represented by configurations $\{e_0\}$ and $\{e_0, e_1\}$, respectively. The concept of configuration of a PES can be formally stated as follows:

Definition 6 (Configuration, Local Conf. and Conf. Extension). Let $\mathcal{E} = (E, \leq, \#, \parallel, \lambda)$ be a PES. A *configuration* C is a set events of $\mathcal{E}$, i.e. $C \subseteq E$, such that the following two conditions hold:

- Given an event $e \in C$, for every e' such that $e' \leq e$ it implies $e' \in C$, and
- For all $e, e' \in C$, it holds $\neg(e \# e')$.

The *local configuration* of event e , denoted $[e]$, refers to the set of events that causally precedes e , i.e. $[e] = \{e' \mid e' \leq e\}$.

We write $C \oplus e$ and call it a *configuration extension*, iff the set $C \cup \{e\}$ is also a configuration.

In the running example, we have that $[e_0] = \{e_0\}$, $[e_1] = \{e_0, e_1\}$, $[e_2] = \{e_0, e_2\}$, so on and so forth. Moreover, we have that the set $\{e_0, e_1, e_2\}$ is a configuration but is not a local one. We will use $\mathcal{F}^\infty(\mathcal{E})$ to represent the (possibly infinite) set of configurations of the event structure $\mathcal{E}$. In case of a PES prefix $\bar{\mathcal{E}}$, we will use $\mathcal{F}(\bar{\mathcal{E}})$ to denote the set of all the configurations of $\bar{\mathcal{E}}$ and $\mathcal{F}_m(\bar{\mathcal{E}})$ the set of maximal configurations with respect to set inclusion.

We write $[e] \uparrow$ to denote the *future* of event e . Clearly, the future of a cutoff event e is isomorphic to the future of its corresponding event f . Relying on such intuition, a “shift” operation over configurations in a net unfolding was introduced [11], that can be used to navigate through configurations and thus reach the states of computation (e.g. a configuration $C \in \mathcal{F}^\infty(\mathcal{E})$) beyond the one explicitly represented in a complete prefix unfolding (e.g. by applying a number of shift operations). For example, the execution of the sequence of tasks A, D from our running example would be associated with the configuration $\{e_0, e_3\}$ over the PES prefix shown in Fig. 9. If we now consider the execution of the sequence of tasks A, D, B , we would need to consider first the configuration $\{e_0, e_3, e_7\}$. Since e_7 is cutoff event and its corresponding event is e_0 we would need to “shift” from $\{e_0, e_3, e_7\}$ to $\{e_0\}$ and only then extend to configuration $\{e_0, e_1\}$.

There are some special cases where the shift operation requires the remapping of some events. The example shown in Fig. 11 is one of those cases. Let us assume that we are analyzing the state of execution represented by configuration $\{e_b, e'_b\}$ and that we want to extend such configuration with event e_c . Since e_c is a cutoff event, we would shift to the corresponding event. In that case, we would need to map e'_b to e_b to reflect fully the isomorphism of the computation. To support such a mapping, we need to identify the isomorphism of events induced by a shift operation, which we would denote $\mathcal{I}_{([e], [f])}$ for a given cc-pair (e, f) . The concrete method to compute such isomorphism is presented in [11]. The following definition formally captures the intuition of the full isomorphic “shift” operation:

Definition 7. Let C and e be respectively a configuration and an event of the PES prefix $\bar{\mathcal{E}}$. The e -shift of C , denoted $\mathcal{S}_e(C)$, is defined as follows:

$$\mathcal{S}_e(C) = \begin{cases} C & \text{if } e \text{ is not a cutoff event} \\ [f] \cup \mathcal{I}_{([e], [f])}(C \setminus [e]) & \text{if } (e, f) \text{ is a cc-pair of } \bar{\mathcal{E}} \end{cases}$$

Consider again the example where we have the configuration $C = \{e_b, e'_b, e_c\}$ from the unfolding in Fig. 11. We want to determine $\mathcal{S}_{e_c}(C)$, and we know that the local configuration of e_c is $[e_c] = \{e_b, e_c\}$. Therefore, we would need to compute the isomorphism of the events $C \setminus [e_c] = \{e'_b\}$, which corresponds with $\{e_b\}$ as mentioned before. Given that e_A is the corresponding event for e_c we have that $\mathcal{S}_{e_c}(C) = [e_A] \cup \{e_b\} = \{e_A, e_b\}$, as expected.

It is therefore by using the shift operation applied over the set of configurations, that the method introduced in [7] extends a given PES prefix into a larger prefix that makes fully explicit every repetitive behavior found in a process model. For convenience, we include an adapted version of this method in Algorithm 1 that serves this purpose. Note that the algorithm keeps track of the configurations that it has visited by means of configuration $\mathcal{C}$ and a multiset $\mathcal{P}$, which we will conjointly refer to as an extended configuration.

Algorithm 1 Identification of elementary pomsets

```

1: procedure EPOMSETS( $\mathcal{E}, \mathcal{C}, \mathcal{P}$ , var  $\mathcal{Z}$ , var  $\mathcal{X}_C$ , var  $\mathcal{X}_L$ , var  $\mathcal{X}_A$ )
2:   PUSH( $\mathcal{Z}, \mathcal{C}$ )
3:   for  $C \oplus e \in \mathcal{F}(\mathcal{E})$  do
4:      $\mathcal{X}_C = \mathcal{X}_C \cup \{(C \oplus e, \mathcal{P} \cup \{e\})\}$ 
5:     if  $e$  is cutoff and  $\mathcal{S}_e(C \oplus e) \in \mathcal{Z}$  then
6:        $\mathcal{X}_L = \mathcal{X}_L \cup \{(C \oplus e, \mathcal{P} \cup \{e\})\}$ 
7:     else if  $\mathcal{S}_e(C \oplus e) \in \mathcal{F}_m(\mathcal{E})$  then
8:        $\mathcal{X}_A = \mathcal{X}_A \cup \{(\mathcal{S}_e(C \oplus e), \mathcal{P} \cup \{e\})\}$ 
9:     else
10:      EPOMSETS( $\mathcal{E}, \mathcal{S}_e(C \oplus e), \mathcal{P} \cup \{e\}, \mathcal{Z}, \mathcal{X}_C, \mathcal{X}_L, \mathcal{X}_A$ )
11:    end if
12:  end for
13:  POP( $\mathcal{Z}$ )
14: end procedure

```

To illustrate how Algorithm 1 works, we will use the set of configurations of the running example, and their configuration extensions. The procedure EPOMSETS takes as parameters the original event structure $\mathcal{E}$, an extended configuration represented by the set $\mathcal{C}$ and the multiset $\mathcal{P}$, a stack $\mathcal{Z}$ and the sets of extended configurations $\mathcal{X}_C$, $\mathcal{X}_L$ and $\mathcal{X}_A$. The procedure is initially called with an extended configuration set based on the empty configurations (both set $\mathcal{C}$ and multiset $\mathcal{P}$). That initial extended configuration corresponds to the left-most node in Fig. 12. Conceptually, EPOMSETS performs a depth-first search traversal over extended configurations by applying the configuration extension (or shift when required) operation. The stack $\mathcal{Z}$ explicitly keeps track of the

Fig. 10. Expanded PES prefix unfolding for PES prefix from Fig. 9.

Fig. 11. Example of isomorphism (a) and its unfolding (b).

traversal. In line 3, EPOMSETS processes each one of the possible configuration extensions. For our running example, there exists only one possible configuration extension: $\{\} \oplus e_0$. Note that in line 4, the tuple $(\{e_0\}, \{\{e_0\}\})$ is added to $\mathcal{X}_C$. In fact, $\mathcal{X}_C$ stores explicitly all the extended extensions. It is also worth noting that the arcs of the graph in Fig. 12 correspond to the subset relation between the configurations. In line 5, the algorithm analyzes the case where a cutoff event and the shift operation over the configuration to be explored has been already visited before, i.e. $S_e(C \oplus e)$ is already in the stack $\mathcal{Z}$. Note that in this case, we are looping back to the entry point of a repetitive behavior, e.g. when we shift from e_4 or e_7 to e_0 in our example.

Note that not all shift operations are necessarily entailing repetitive behavior. That is, there are cases where the shift operation will move to another branch in the unfolding to progress forwards. However, when the configuration derived with a shift operation is already stored in $\mathcal{Z}$ we can be sure that we have identified an (elementary) cycle (see [7]). In line 6, we add the extended configuration $(C \oplus e, \mathcal{P} \cup \{e\})$ to $\mathcal{X}_L$, which clearly stores the extended configurations characterizing that loop back, defining thus the inherent repetitive behavior. With abuse of notation, we will use $\text{entry}(L)$ to denote the extended configuration that corresponds to the entry point to looping behavior induced by the extended configuration $L \in \mathcal{X}_L$. In lines 7 and 8, the algorithm checks if we have reached a maximal configuration in which case we add it to $\mathcal{X}_A$, which stores the extended configurations that reach a state where the computation stops.

At the end of Algorithm 1, we would have navigated all the (elementary) cycles and collected all the extended configurations $\mathcal{X}_C$, the set $\mathcal{X}_L$ of extended configurations defining repetitive behavior, and the set $\mathcal{X}_A$ of extended configurations that correspond to maximal behavior. In such behavior, we stopped the cycles without letting them reach a maximal behavior. However, it is possible to complete the set of extended configurations to represent one full repetition of every (elementary) cycle by combining the information of the three sets $\mathcal{X}_C$, $\mathcal{X}_L$ and $\mathcal{X}_A$ as follows:

Definition 8. Let $C = (C, \mathcal{P})$ and $C' = (C', \mathcal{P}')$ be extended configurations. The union of extended configurations C and C' is defined as $C \uplus C' = (C \cup C', (\mathcal{P} \setminus C) \cup \mathcal{P}')$.

Table 1
CPES events.

$\mathbf{e} \in E_{\mathcal{V}}$	$\lambda_{\mathcal{V}}$
$\{A_1\}, \{A_2\}, \{A_1, A_2\}$	A
$\{B_1\}, \{B_2\}, \{B_1, B_2\}$	B
$\{C_1\}, \{C_2\}, \{C_1, C_2\}$	C
$\{X_1\}$	X
$\{Y_2\}$	Y

the set of extended configurations $C \in \mathcal{X}_C$ such that $\text{entry}(L) \subset C$ and $C \subset L$.

4. Creating a set of common behavior relations

To capture the generic behavior between multiple business processes, we merge the behavior of each input process by combining each input PES prefix unfolding into a *Compound Prime Event Structure* (CPES). A CPES can be formally defined as follows:

Definition 9 (Compound Prime Event Structure). Let $\mathcal{V} = \{\hat{\mathcal{E}}(N_0), \dots, \hat{\mathcal{E}}(N_n)\}$ be a set of PES prefix unfoldings. A compound prime event structure over $\mathcal{V}$ is a quadruple $\mathcal{E}_{\mathcal{V}} = (E_{\mathcal{V}}, \xrightarrow{t}, \rightarrow, \lambda_{\mathcal{V}})$, where:

- $E_{\mathcal{V}} = \left\{ \mathbf{e} \subseteq \bigcup_{\hat{\mathcal{E}}(N_i) \in \mathcal{V}} E_i \mid \mathbf{e} \neq \emptyset \wedge \forall e, e' \in \mathbf{e} \implies \lambda(e) = \lambda(e') \right\}$ is the combined set of events, with $\hat{\mathcal{E}}(N_i) = (E_i, \leq_i, \#_i, \lambda_i)$,
- $\xrightarrow{t} = \{<_d, \leq, \#, \parallel\}$ defines the types of relations, where:
 - $\mathbf{e} <_d \mathbf{e}'$ denotes a *direct causality* relation, such that for a PES prefix unfolding $\hat{\mathcal{E}}(N_i) : e_m \in \mathbf{e} \cap E_i \wedge e_n \in \mathbf{e}' \cap E_i \wedge e_m < e_n \wedge \forall e \in [e_n] \setminus [e_m] \implies \lambda(e) = \tau$
 - $\mathbf{e} \leq \mathbf{e}'$ denotes a causality relation, i.e. the transitive closure of $<_d$
 - $\mathbf{e} \# \mathbf{e}'$ denotes a *conflict* relation, such that for a PES prefix unfolding $\hat{\mathcal{E}}(N_i) : e_m \in \mathbf{e} \cap E_i \wedge e_n \in \mathbf{e}' \cap E_i \wedge e_m \# e_n$,
 - $\mathbf{e} \parallel \mathbf{e}'$ denotes a *concurrent* relation, such that for a PES prefix unfolding $\hat{\mathcal{E}}(N_i) : e_m \in \mathbf{e} \cap E_i \wedge e_n \in \mathbf{e}' \cap E_i \wedge e_m \parallel e_n$,
- $\rightarrow$ is the set of relations existing between all events $E_{\mathcal{V}} \times \xrightarrow{t} \times E_{\mathcal{V}}$,
- $\lambda_{\mathcal{V}}$ is a labeling function over $\mathbf{e} \in E_{\mathcal{V}}$ such that $\lambda_{\mathcal{V}}(\mathbf{e}) = \lambda_i(e)$ when $\mathbf{e} \cap E_i = \{e\}$ of the PES prefix unfolding $\hat{\mathcal{E}}(N_i)$.

A CPES $\mathcal{E}_{\mathcal{V}}$ over a set of PES prefix unfoldings $\mathcal{V} = \{\mathcal{E}_0, \dots, \mathcal{E}_n\}$ combines the set of events and relations from all PES prefix unfoldings.

Consider, for instance, the two variants shown in Fig. 13. In Variant 2, X has been replaced by Y. As such, the combined set of events $E_{\mathcal{V}}$ and the corresponding label $\lambda_{\mathcal{V}}$ can be represented as shown in Table 1. Note that the set $\mathbf{e}$ is maximal and uses all required subsets of individual events. The necessity for that becomes clear when obtaining the set of relations $\rightarrow$ between events. Despite involving the same activities (w.r.t. their label), some relations hold for both variants and some for only a single variant. The resulting set of events (given in Box 1) is, as a consequence, different.

The relations between two events $\mathbf{e}$ and $\mathbf{e}'$ enumerated above can be summarized using their labels, which results in a set of relations that exist between pairs of events with labels $\lambda(\mathbf{e})$ and $\lambda(\mathbf{e}')$ as shown in Table 2.

We define an additional function, that allows us to retrieve the set of input event structures in which a certain relation occurs:

Definition 10. $\mathcal{V} \mid_{\mathcal{R}} (\mathbf{eRf}) = \{\hat{\mathcal{E}}(N_i) \mid \exists e \in \mathbf{e}, f \in \mathbf{f} : eRf \wedge \mathbf{eRf} \in \rightarrow\}$.

Table 2
Relations between all events $\rightarrow$.

	A	X	Y	B	C
A	$\{\parallel\}$	$\{<_d, \leq\}$	$\{<_d, \leq\}$	$\{\leq\}$	$\{\leq\}$
X	$\{>_d, \geq\}$	$\{\parallel\}$	$\{\}$	$\{<_d, \leq\}$	$\{\leq\}$
Y	$\{>_d, \geq\}$	$\{\}$	$\{\parallel\}$	$\{<_d, \leq\}$	$\{\leq\}$
B	$\{\geq\}$	$\{>_d, \geq\}$	$\{>_d, \geq\}$	$\{\parallel\}$	$\{<_d, \leq\}$
C	$\{\geq\}$	$\{\geq\}$	$\{\geq\}$	$\{>_d, \geq\}$	$\{\parallel\}$

For example, $\mathcal{V} \mid_{\mathcal{R}} (\{A_1\} <_d \{X_1\}) = \{\hat{\mathcal{E}}(N_1)\}$. That is, the direct causality between A and X occurs in $\hat{\mathcal{E}}(N_1)$ only. Similarly, $\mathcal{V} \mid_{\mathcal{R}} (\{A_1, A_2\} \leq \{B_1, B_2\}) = \{\hat{\mathcal{E}}(N_1), \hat{\mathcal{E}}(N_2)\}$, indicating that the transitive causality between A and B occurs in both input event structures.

In addition, we define a function that returns the set of input event structures in which a given label occurs.

Definition 11. $\mathcal{V} \mid_{\lambda} (l) = \{\hat{\mathcal{E}}(N_i) \mid \exists e \in E_i : \lambda_i(e) = l\}$.

For example, we can obtain the set of input event structures that contain an event with the same label as X_1 using $\mathcal{V} \mid_{\lambda} (\lambda(X_1)) = \{\hat{\mathcal{E}}(N_1)\}$. Similarly, the set of input event structures that contain an event with the same label as C_1 can be obtained by $\mathcal{V} \mid_{\lambda} (\lambda(C_1)) = \{\hat{\mathcal{E}}(N_1), \hat{\mathcal{E}}(N_2)\}$.

5. Declarative variability rules

To represent the behavior of a family of related business processes as a set of declarative variability rules, we use a branching-time temporal logic. Branching-time temporal logics specify events over future states on branching paths, or tree-like structures, where each branch represents a possible execution path. As such, it is very suitable for defining possible execution paths by specifying temporal relations between events. One of the most notable branching-time temporal logics is Computation Tree Logic⁴ (CTL) [16], which is often used with formal verification. CTL is especially useful when considering *structural* properties [17] over the different possible branching constructs contained in business process models.

Definition 12 (CTL Syntax). The language of well-formed CTL formulae is generated by the following grammar, assuming $p \in AP$:

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

CTL is equipped with four *temporal operators*:

- $\neg X\phi$ Nexttime: ϕ has to hold at the next state.
- $\neg G\phi$ Globally: ϕ has to hold on every state of the subsequent path.
- $\neg F\phi$ Future: ϕ eventually has to hold in a future state.
- $\neg [\phi \text{ U } \phi']$ 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 whether some or all branches possess properties starting at the current state. CTL defines the following operators over paths:

⁴ The semantics of CTL are defined on a Kripke structure, which is shown formally in Appendix A. The class of CTL formulas used throughout this paper require a Kripke structure with a business process specific labeling known as a transition graph [15,1]. A transition graph is a Kripke structure obtained from a Petri net which maintains parallel and local next behavior (i.e., the next activity occurrence within a single branch) of business process models.

Fig. 13. Linear variants.

$\{A_1\} <_d \{X_1\}$	$\{A_1, A_2\} \leq \{C_1, C_2\}$	$\{Y_2\} \leq \{B_2\}$	$\{C_1\} \parallel \{C_1\}$
$\{A_2\} <_d \{Y_2\}$	$\{X_1\} \leq \{C_1\}$	$\{B_1, B_2\} \leq \{C_1, C_2\}$	$\{A_2\} \parallel \{A_2\}$
$\{X_1\} <_d \{B_1\}$	$\{Y_2\} \leq \{C_2\}$	$\{A_1\} \parallel \{A_1\}$	$\{Y_2\} \parallel \{Y_2\}$
$\{Y_2\} <_d \{B_2\}$	$\{A_1\} \leq \{X_1\}$	$\{X_1\} \parallel \{X_1\}$	$\{B_2\} \parallel \{B_2\}$
$\{B_1, B_2\} <_d \{C_1, C_2\}$	$\{A_2\} \leq \{Y_2\}$	$\{B_1\} \parallel \{B_1\}$	$\{C_2\} \parallel \{C_2\}$
$\{A_1, A_2\} \leq \{B_1, B_2\}$	$\{X_1\} \leq \{B_1\}$		

Box I.

- $A\psi$ 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.

Given a CPES $\mathcal{E}_V = (E_V, \xrightarrow{t}, \Rightarrow, \lambda_V)$ over $V = \{\hat{E}(N_0), \dots, \hat{E}(N_n)\}$ with $\hat{E}(N_i) = (E_i, \leq_i, \#_i, \lambda_i)$, we define the declarative variability rules as seven types of CTL formulae that together describe a family of related business processes. Since CTL reasons over properties in *all* or *some* paths, we first specify upper bound rules for *all* paths starting at, and leading to, events. These two types, *immediate response* (Definition 13) and *precedence* (Definition 14), define the possible execution paths that may occur by enumerating all possible direct causality relations from each event in maximal sets of events with equal labels.

Definition 13 (*Immediate Response*). Immediate response rules, $AG(\lambda_V(P) \Rightarrow \phi)$, are CTL formulae such that:

$$\phi = \begin{cases} A[(\lambda_V(P) \vee \tau) \cup (\bigvee_{q \in Q} \lambda_V(q))] & \text{iff } S = \emptyset \\ AF(\bigvee_{q \in Q} \lambda_V(q)) & \text{otherwise} \end{cases}$$

- $P \subseteq E_V$ where for all $p, p' \in P : \lambda_V(p) = \lambda_V(p')$ and P is maximal with respect to set inclusion,
- $Q = \{q \mid q \in E_V \wedge \exists p \in P : p <_d q\}$ holds all direct causal events, and
- $S = \{s \mid s \in E_V \wedge \exists p, p' \in P, q \in Q, r, r' \in E_V : \lambda_V(p) <_d q \cap \lambda_V(r) <_d s \cap \lambda_V(p') \parallel r' \neq \emptyset \wedge \lambda_V(r) = \lambda_V(r') \wedge \lambda_V(s) = \lambda_V(q) \wedge \lambda_V(p') \neq \lambda_V(r')\}$ holds all synchronizing events.

For example, consider activity A of Fig. 13, which exists in both variants but is followed by different activities. To create an *immediate response* rule for this activity, the set P consists of the maximal set of events in the CPES with the label A, i.e. $\{A_1\}, \{A_2\}, \{A_1, A_2\}$.⁵ With abuse of notation, we use $\lambda_V(P)$ to denote the label of each $p \in P$. The set Q , in turn, holds all events for which there exist a direct causal relation with any event in P . Since $\{A_1\} <_d \{X_1\}$ holds for Variant 1, and $\{A_2\} <_d \{Y_2\}$ holds for Variant 2, the set Q consists of $\{X_1\}, \{Y_2\}$. As a result, we can deduce the following *immediate response* rule:

$$AG(A \Rightarrow A[(A \vee \tau) \cup (X \vee Y)])$$

Although this rule works well for verifying the order of events in sequences, the order of events cannot be guaranteed in case of synchronization between concurrent paths. As such, events where parallel paths synchronize may be waiting for one of the branches to complete when all other concurrent branches have already

completed. For example, consider activity B of Fig. 14, which is followed by activity D in both variants.

In this example, activity D may be waiting for activity C, even though B already executed and B is direct causal to D. In this case, an *immediate response* relation as specified above would not hold for any of the already completed branches, as the synchronizing event is not the immediate response event. Instead, an activity in the uncompleted branch is the event that will be executed next (activity C in this example).

Therefore, the *immediate response* type where parallel paths synchronize requires an alternate (eventual) representation to allow for possible parallel branches to finish. Here, the set P consists of $\{B_1\}, \{B_2\}, \{B_1, B_2\}$. Since $\{B_1, B_2\} <_d \{D_1, D_2\}$ holds, the set Q consists of $\{D_1, D_2\}$. In this case, however, there is parallel behavior and Q includes *synchronizing events*. As such, we introduce a set S to account for the synchronization of different parallel branches. To find the synchronizing events for Variant 1, we know that $\{B_1, B_2\} <_d \{D_1, D_2\}$, $\{X_1\} <_d \{D_1\}$, and $\{B_1\} \parallel \{X_1\}$ hold. As a result, we can find $p = \{B_1, B_2\}$, $p' = \{B_1\}$, $q = \{D_1, D_2\}$, $r = \{X_1\}$, and $r' = \{X_1\}$, giving us $s = \{D_1\}$ as synchronizing event of Variant 1. Similarly, for Variant 2 we find $s = \{D_2\}$. As a result, the set S consists of $\{D_1\}, \{D_2\}$. Using P, Q and S , we can deduce the following *immediate response* rule:

$$AG(B \Rightarrow AF D)$$

Definition 14 (*Precedence*). Precedence rules, $\neg E[\neg(\bigvee_{q \in Q} \lambda_V(q)) \cup \lambda_V(P)]$, are CTL formulae such that:

- $P \subseteq E_V$ where for all $p, p' \in P : \lambda_V(p) = \lambda_V(p')$ and P is maximal with respect to set inclusion, and
- $Q = \{q \mid q \in E_V \wedge \exists p \in P : q <_d p\}$ holds all direct causal events.

For example, consider a *precedence* rule for activity B of Fig. 13. In this case, the set P consists of $\{B_1\}, \{B_2\}, \{B_1, B_2\}$. Since $\{X_1\} <_d \{B_1\}$ holds for Variant 1, and $\{Y_2\} <_d \{B_2\}$ holds for Variant 2, the set Q consists of $\{X_1\}, \{Y_2\}$. As a result, the following *precedence* rule is created:

$$\neg E[\neg(X \vee Y) \cup B]$$

Next, we specify the lower bound rules. These two types, *exists immediate response* (Definition 15) and *exists eventual response* (Definition 16), define the mandatory responses between events that must occur in some paths by iterating over pairs of event for which causality relations exists in all business processes in the family.

Definition 15 (*Exists Immediate Response*). Exists immediate response rules, $AG(\lambda_V(p) \Rightarrow E[(\lambda_V(p) \vee \tau) \cup \lambda_V(q)])$, are CTL formulae such that:

⁵ We use subscript to denote which event occurred in which variant.

Fig. 14. Parallel variants.

- $p, q \in E_V : p <_d q$, i.e., direct causal relations exist, and
- $\mathcal{V}_\lambda(\lambda_V(p)) = \mathcal{V}_\lambda(p <_d q)$, i.e., for all PES prefix unfoldings that contain p , $p <_d q$ holds.

For example, consider an *exists immediate response* rule for activities X and B of Fig. 13. In this case, it is clear that $\{X_1\} <_d \{B_1\}$ holds, but it is also clear that there exists no such relation for B of Variant 2. This is true, because X only exists in Variant 1. However, since we define the *exists immediate response* rule using an implication, we only require the direct causality relation to hold for those variants in which X exists. That is, we require the set of PES prefix unfoldings in which X exists (i.e., $\mathcal{V}_\lambda(X)$) to be equal to the set for which the relation $\{X_1\} <_d \{B_1\}$ holds (i.e., $\mathcal{V}_\lambda(\{X_1\} <_d \{B_1\})$). Here this condition holds, causing the following *exists immediate response* rule to be deduced:

$$AG(X \Rightarrow E[(X \vee \tau) \cup B])$$

Definition 16 (*Exists Eventual Response*). Exists eventual response rules, $AG(\lambda_V(p) \Rightarrow EF\lambda_V(q))$, are CTL formulae such that:

- $p, q \in E_V : p \leq q \wedge \neg(p <_d q)$, i.e., causal relations exist,
- $\mathcal{V}_\lambda(\lambda_V(p)) = \mathcal{V}_\lambda(p \leq q)$, i.e., for all PES prefix unfoldings that contain p , $p \leq q$ holds, and
- $\nexists q', r, r' \in E_V : (\mathcal{V}_\lambda(\lambda_V(p)) = \mathcal{V}_\lambda(p \leq r)) \wedge (\mathcal{V}_\lambda(\lambda_V(r')) = \mathcal{V}_\lambda(r' \leq q')) \wedge \lambda_V(r) = \lambda_V(r') \wedge \lambda_V(q) = \lambda_V(q')$, i.e., there is no r that can create two rules which together imply this rule.

For example, consider an *exists eventual response* rule between activities A and B of Fig. 13. It is clear that $\{A_1, A_2\} \leq \{B_1, B_2\}$ holds, and includes all PES prefix unfoldings that contain A, while $\{A_1, A_2\} <_d \{B_1, B_2\}$ does not hold. To prevent the creation of large amounts of redundant rules, an additional condition is included. This condition makes sure that we only create exists eventual response rules when the rule cannot be described by several similar rules. That is, this condition requires that there is no r for which $\{A_1, A_2\} \leq r$ and $r \leq \{B_1, B_2\}$ holds in all PES prefix unfoldings that contain A and $\lambda_V(r)$, respectively. In this case, the condition holds because no lower bound rule can be specified from activity A to a consecutive activity, since it is followed immediately by different activities in the two variants. As a result, the following *exists eventual response* rule can be deduced:

$$AG(A \Rightarrow EFB)$$

Next, we specify the *concurrency* rules (Definition 17). This type defines the possible simultaneous execution of activities by iterating over pairs of events for which concurrent relations exist in all business processes in the family. Note that the *concurrency* rule requires a possible concurrent execution. This requirement is strong enough to enforce their existence in separate parallel branches structurally.

Definition 17 (*Concurrency*). Concurrency rules, $EF(\lambda_V(p) \wedge \lambda_V(q))$, are CTL formulae such that:

- $p, q \in E_V : \lambda_V(p) \neq \lambda_V(q) \wedge \mathcal{V} = \mathcal{V}_\lambda(p \parallel q)$, i.e., $p \parallel q$ holds always between disjointed events.

For example, consider a *concurrency* rule for activities B and C of Fig. 14. It is clear that $\{B_1, B_2\} \parallel \{C_1, C_2\}$ holds and includes all PES prefix unfoldings. In addition, $\lambda_V(\{B_1, B_2\}) \neq \lambda_V(\{C_1, C_2\})$ also holds. We include this additional requirement, because events may be concurrent with itself in PES prefix unfoldings. Note that these requirements are not true for the activities B and X or B and Y, and that concurrency rules cannot be created for those activities. As a result, only the following *concurrency* rule is created:

$$EF(B \wedge C)$$

Next, we specify the *conflict* rules (Definition 18). This type excludes activities from executing in the same path by iterating over pairs of events for which conflict relations exist for all business processes in the family that contain both events. Although the *conflict* type only defines a conflict in a single direction (i.e., from p to q), the symmetry of the conflict relation is maintained within the CPES, which ensures the inclusion of the specification in the other direction.

Definition 18 (*Conflict*). Conflict rules, $AG(\lambda_V(p) \Rightarrow AG \neg \lambda_V(q))$, are CTL formulae such that:

- $p, q \in E_V : p \# q$, i.e., conflict relations exist, and
- $\mathcal{V}_\lambda(\lambda_V(p)) \cap \mathcal{V}_\lambda(\lambda_V(q)) = \mathcal{V}_\lambda(p \# q)$, i.e., for all PES prefix unfoldings that contain p and q , $p \# q$ holds.

For example, consider possible *conflict* rules for activities B and X of Fig. 15. It is clear that $\{B_2\} \# \{X_2\}$ and $\{X_2\} \# \{B_2\}$ hold, but also that such a conflict relation does not exist for Variant 1. This is true, because X only exists in Variant 2. However, we only require the conflict relation to hold for those variants in which both B and X exist. Since this is only the case for Variant 2, the following *conflict* rules are created:

$$AG(B \Rightarrow AG \neg X)$$

$$AG(X \Rightarrow AG \neg B)$$

Finally, we specify the *cyclic* rule (Definition 19). This type defines repetition of activities in the form of cycles that hold for all business processes in the family by requiring the activity to sometimes hold, then release, and hold again. Note that this type only specifies repetition of an activity itself because the *exists immediate response* and *exists eventual response* types already implicitly define repetition between different activities in cycles.

Definition 19 (*Cyclic*). Cyclic rules, $AG(\lambda_V(p) \Rightarrow E[\lambda_V(p) \cup E[\neg \lambda_V(p) \cup \lambda_V(p)]])$, are CTL formulae such that:

- $p \in E_V \wedge \forall \hat{e}(N_i) \in \mathcal{V}_\lambda(\lambda_V(p)), \exists e \in p : e \in \mathcal{L}(\hat{e})$, i.e., for all PES prefix unfoldings that contain p , it is in a cycle.

For example, consider a *cyclic* rule for activity B of Fig. 16. In this case, p can be either $\{B_1\}$, $\{B_2\}$, or $\{B_1, B_2\}$. However, since we require an event e in p for which e is contained in the set of events in the loop $\mathcal{L}(\hat{e})$ of each PES prefix unfolding that contains the label of p , only the latter set will produce a cyclic rule. That is, the set $\{B_1, B_2\}$ contains B_1 which is in the loop of Variant 1 and B_2 which is in the loop of Variant 2. As a result, the following *cyclic* rule is created:

$$AG(B \Rightarrow E[B \cup E[\neg B \cup B]])$$

Fig. 15. Exclusive variants.

Fig. 16. Cyclic variants.

The resulting set of variability rules, denoted by V , is considered to be the basic set of rules required to define a coherent, yet variable, family of business processes. Other, more strict, rules can, however, easily be included on a per application case basis. For example, a rule could be included to enforce the exclusion of unrelated events, i.e., the exclusion of events that did not appear together in any PES prefix unfolding. In this way, rules can be introduced for any combination of relations and existence of events with certain labels. However, as additional rules are introduced, the variability rules become more strict, allowing a smaller range of variants. The decision of introducing additional rules is, therefore, left to be considered per application case.

6. Evaluation

We demonstrate the presented approach by conducting an exploratory evaluation on different sets of real-life business process variants. First, we use the real-life process models from eGovernment presented in Section 2 to assess the performance and number of produced variability rules for each type of rules. We subsequently illustrate the potential applications of the approach, by first combining two variants and then introducing a fourth eGovernment variant and assessing their conformance with the variability specification obtained from the first three variants. Finally, we perform a number of experiments on a large data set containing different sets of process variants, to explore the relation between the type and amount of rules for specifications with increasing amounts input variants having an increasing complexity.

6.1. Tool implementation

The automated generation of specifications is realized both as a standalone Java package called ProVariant⁶ and as a plugin in the Apromore advanced business process analytics platform⁷. For the evaluation, we use the plugin in Apromore, as it features a convenient graphical user interface.

Apromore is an open-source and extensible online platform, comprising state-of-the-art capabilities for managing and analyzing large process model collections. The presented approach complements a wide range of existing capabilities provided by Apromore, like process model merging, simulation and similarity search [18]. The plugin takes as input a set of Petri net models in PNML format.⁸ Its output is the generated set of variability rules, as formally defined in Definitions 13 to 19.

Although the approach requires PNML models as input, the modeling languages supported are BPMN, CPF, EPCs, Petri nets and YAWL as Apromore offers conversions from any of these formats

Table 3

Output of variability rules for each individual process.

Models	#	Im.Resp.	Prec.	∃ Im.Resp.	∃ Evt.Resp.	Conf.	Conc.	Cyc.	Total
A	CTL	20	20	31	0	54	0	9	134
	E	21	21	21	0	10	0	9	21
B	CTL	23	23	38	0	88	0	8	180
	E	24	24	24	0	21	0	8	24
C	CTL	28	28	55	0	140	0	10	261
	E	29	29	29	0	26	0	10	29

to the required PNML format. As such, the implementation of the specification generation approach presented here is independent of the process model input-format used.

6.2. Variability rules

To evaluate the variability rules generated by our method, we used the model of Municipality A as the general case, where Municipality B and Municipality C are variants. In Table 3, an overview is provided of the rules generated for the individual models. For each model, all existing response relations can be covered by *exists immediate response*, as there is only one model involved and, hence, no variability. Consequently, there are no *exists eventual response* rules for each individual model. Furthermore, none of the models have concurrent activities. Municipality C has the most activities, of which the majority is involved in (multiple) loops. This is represented by the much higher number of *exists immediate response* and *conflict* rules, particularly when compared to the involved events.

Subsequently, we have created three distinct variability rule sets. The first rule set combines Municipality A and Municipality B, which show low variability (i.e., the amount and complexity of the differences between the two models is relatively low). The second rule set concerns Municipality A and Municipality C, which show high variability (i.e., the amount and complexity of the differences between the two models is relatively high). The third rule set combines all three models. Table 4 provides an overview of the sets of variability rules produced for each combination of models.

It is easy to see that the amount of *immediate response* and *precedence* rules is dependent on the amount of different activities throughout all models. The amount of *conflict* rules are dependent on the model with the least amount of conflicts, as those are restrictive and should adhere to the least restrictive model (Municipality A).

Clearly, an increasing variability and complexity results in a gradually increasing number of rules. Furthermore, Municipality C differs much more from Municipality A than Municipality B does, which is clearly shown by the number of rules resulting from the case featuring Municipality A and C. The amount of rules here are almost identical to the amount of rules required for the third case involving all models. Consequently, it can be observed that most of

⁶ Available at <http://www.hgroefsema.nl>

⁷ <http://apromore.org/>

⁸ For a detailed description of the PNML standard, see <http://www.pnml.org/>

Table 4

Output of variability rule sets composed of different processes.

Models	#	Im.Resp.	Prec.	$\exists$ Im.Resp.	$\exists$ Evt.Resp.	Conf.	Conc.	Cyc.	Total
1 A,B	CTL	23	23	30	1	72	0	6	155
	E	24	24	24	2	13	0	6	24
2 A,C	CTL	29	29	37	2	126	0	8	231
	E	30	30	29	4	19	0	8	30
3 A,B,C	CTL	30	30	35	2	124	0	5	226
	E	31	31	28	4	20	0	5	31

Table 5

Amount of overlapping rules between different variability rule sets.

Rule sets	Im.Resp.	Prec.	$\exists$ Im.Resp.	$\exists$ Evt.Resp.	Conf.	Conc.	Cyc.	Total
1 vs. 2	42%	38%	69%	67%	71%	N.A.	71%	62%
1 vs. 3	53%	49%	74%	69%	73%	N.A.	73%	67%
2 vs. 3	68%	64%	94%	100%	98%	N.A.	77%	88%

Municipality B fits within the behavior of both Municipality A and Municipality C, as the addition of Municipality B to the variability case hardly changes the required amount of rules.

Table 5 shows the percentage of overlapping rules between a pair of rule sets. Rule sets 1 and 2 show a total overlap of 62%, i.e., 62% of the rules are common between both rule sets. This does also take into account partial overlap between rules: when additional events are included as a possible preceding or responding event, the existing rules are extended without requiring additional rules.

Rule sets 1 and 3 show already a slightly higher overlap (67%), as rule set 3 also includes B. It can be observed that this has a significant impact on the amount of overlapping immediate response and precedence rules. The differences between rule set 2 and 3 are significantly smaller with a total overlap of 88%, which can be explained by the inclusion of the most deviant municipality (i.e. C) in both rule sets.

This strongly supports the case of the necessity of such variability rule sets, as *roughly* the processes are similar across different municipalities, but differ with respect to details and municipality specific policies. Consequently, a generic variability rule set (like rule set 3) allows for support of all variants, while maintaining the common control-flow.

6.3. Verification of a combined process variant

In addition to providing a declarative variability rule set for a family of process variants, the approach presented in this paper allows us to verify whether certain process variants are compliant with the variability rules and, as such, check the conformance of a new process variant to an existing variability rule set. In [1], an approach is presented to verify the compliance of an existing business process with respect to a set of rules specified using the temporal logic CTL. The input consists of a business process model and a set of CTL rules. The output contains an overview of all rule evaluations (i.e. true or false) along with a detailed counter example describing the corresponding faulty path for each rule that evaluates to false.

The verification approach is implemented both as a standalone Java package (Available online at <http://www.hgroefsema.nl>) and as a plugin in Apromore. The CTL rules required for the verification plugin are compatible with the CTL rules produced in this paper and can, as such, be used as a toolchain together with the approach presented here.

Let us assume a scenario where a municipality wants to use a part of the process as specified by Municipality A and part of the process as specified by Municipality C. This combined variant is graphically represented in Fig. 17. Using the variability rules as

Fig. 17. Municipality C combined with A.

obtained in Section 6.2, we employ the method presented in [1] to verify whether this combined variant conforms to the variability rules obtained from municipalities A through C.

The variability rules obtained from municipalities A to C were imported in Apromore, together with the new combined process model. Subsequently, the model shown in Fig. 17 was verified against the CTL rules. The verification revealed that the behavior allowed by the model in Fig. 17 is supported by the rules obtained from rule set 3 above.⁹ That is, each of the 212 rules holds for the combined process model.

This result was expected, as the combined model does not contain any *new* behavior. All paths shown in Fig. 17 already occur in Municipality A (Fig. 3) or Municipality C (Fig. 5) and are, as such, supported by the rules. Consequently, the use of declarative variability rules allows to define the “boundaries” of the behavior as specified in the input variants, and allows all behavior “in between”.

6.4. Verification of a new process variant

Now let us assume a different variant representing the business process as executed by municipality D, which is shown graphically

⁹ Under condition of some fairness assumptions on loops.

It is clear that, with proper interpretation, the use of a declarative variability rule set results in very precise feedback on the parts of the new process variant that are not conformant with the variability rule set: it not only shows which events are misplaced, inserted or missing, it also indicates which paths are invalid for the new process variant.

When starting with the most complex model and subsequently adding the less complicated models (as opposed to the other way around), the initial amount of rules is higher, but approaches the same complexity when adding more/all models.

¹⁰ <http://www.se-rwth.de/materials/semdiff/>.

Table 6

Overview of rules in the data set, sorted on increasing complexity.

		# Included variants											
		1	2	3	4	5	6	7	8	9	10	11	12
Anonymous	Event labels	17	22	28									
	Total rules	34	42	58									
Bank	Event labels	17	19										
	Total rules	34	31										
Big	Event labels	15	15										
	Total rules	35	35										
Hiring	Event labels	14	18	18	19								
	Total rules	22	26	24	23								
IBM	Event labels	30	33	49	50	103	105	137	139	155	155	164	164
	Total rules	84	99	165	165	296	299	376	379	413	413	443	443
KGE	Event labels	22	29	36									
	Total rules	51	80	121									
Municipality	Event labels	25	32	35	40	42							
	Total rules	134	120	143	166	212							
Nokia	Event labels	16	20	28	28								
	Total rules	36	44	60	60								
Proj	Event labels	10	10	13									
	Total rules	20	20	19

Fig. 19. Rules vs. labels for variants of increasing complexity.

Table 7
Overview of rules in the data set, sorted on decreasing complexity.

		# Included variants											
		1	2	3	4	5	6	7	8	9	10	11	12
Anonymous	Event labels	22	24	28									
	Total rules	34	42	58									
Bank	Event labels	15	19										
	Total rules	34	31										
Big	Event labels	15	15										
	Total rules	35	35										
Hiring	Event labels	15	16	19	19								
	Total rules	22	26	24	23								
IBM	Event labels	15	18	35	36	73	75	121	123	131	133	161	164
	Total rules	84	99	165	165	296	299	376	379	413	413	443	443
KGE	Event labels	27	33	36									
	Total rules	51	80	121									
Municipality	Event labels	38	38	40	42	42							
	Total rules	261	258	245	217	212							
Nokia	Event labels	24	24	26	28								
	Total rules	36	44	60	60								
Proj	Event labels	12	13	13									
	Total rules	20	20	19									

An overview of the total rules and the event labels for models ordered on decreasing complexity is provided in Table 7. The impact of the most complex model is particularly evident for the Municipality models, where the total amount of rules is even decreasing when adding more models. For this specific set of variants, the difference in complexity between the most complex variant and the least complex variant is by far the largest among the different models evaluated in this paper. As a consequence, the rule set containing a single variant (i.e. the most complex variant) requires the highest amount of rules to describe the behavior of the model. Any subsequent variant added to the rule set reduces the amount of rules, as it relaxes a number of the constraints imposed by the first variant.

The relations between individual types of rules and the amount of labels is illustrated graphically for all data sets in Fig. 20. The trends that can be identified here confirm the observations made based above on the analysis for increasing complexity, as the tendency for the amount of rules to decrease relative to the amount of labels involved is even more pronounced.

6.6. Performance and computational complexity

For each variability case, the execution times required to obtain the rule sets were captured. The analysis was executed using a laptop with Intel i7 2.5 GHz, running JVM 8 with 16 GB of allocated memory. In all cases, the generation of the declarative variability rules completed within 25 ms.

The theoretical complexity of the entire method can be derived as follows. The complexity of the transformation step of each Petri net is defined by $\mathcal{O}(\left(\frac{|B|}{\zeta}\right)^\zeta)$, where B is the set of places (conditions) of the unfolding and ζ is the maximal size of the presets of the transitions in the original net [20]. The subsequent transformation of the complete prefix unfolding into an event structure is linear time using the concurrency relation during the computation of the unfolding. Finally, the complexity of the construction of the compound behavior relation sets from Definition 9 is defined by $\mathcal{O}(|E_V|^2)$. As such, the total complexity is dominated by the transformation step from each Petri net to its respective unfolding and subsequent event structure, and, therefore, the amount of input processes involved in the set of variability rules.

7. Related work

Business process variability is closely related to design-time process *adaptability* and runtime process *flexibility*, which both support process change [21]. Existing approaches that offer process change can be subdivided into those allowing change within imperatively specified business processes and those allowing change through a declarative approach [22–24].

When offering design-time change within imperatively specified business processes, many existing approaches apply principles directly from variability as used within software product lines – such as feature modeling and variation points [25]. A software product line consists of a family of closely related software products with a single generic implementation. Differences between each product within the product line are described using feature models. When creating a variant, the relevant features are selected and integrated at, so called, variation points within the generic implementation. When applying this principle to business process adaptability, the generic implementation is provided by an imperatively specified business process with included variation points. Approaches using these principles include [26–37].

On the other hand, Hallerbach et al. [38] employs the same principles to offer change during process enactment. Of course, a single generic model incorporating all possible variations can contain configurations which lead to unsound processes. To address this problem, van der Aalst et al. [39] propose a verification approach which is able to characterize all feasible configurations at design-time, while in [40], the authors propose configurations that are able to maintain correctness.

In order to obtain generic models, [41–45] propose process merger. When applying this approach, variants are merged into a single generic model using a number of different techniques such as a new temporal process logic TPL, or CoSeNets. Likewise, Buijs et al. [46] propose and evaluate four merging techniques to describe a family of business process variants using configurable process models.

Alternatively, van der Aalst and Basten [47] utilize principles from object-oriented programming languages to define process inheritance. Inheritance is a mechanism which allows a subclass to inherit features from a superclass. When applied to business processes, inheritance defines a bisimilarity relation over two process models. A process model is a subclass of another process model when the subclass and superclass are bisimilar under certain conditions. Milani et al. [48], on the other hand, propose a

Fig. 20. Rules vs. labels for variants of decreasing complexity.

decomposition based method using sub-processes which decides which parts should be modeled together, and which should not.

Design-time change within imperatively specified business processes, however, require all possible features to be modeled (and therefore known) in advance. In addition, some features may have relations with other features (e.g., be exclusive or pre-required) that must be modeled. Assy et al. [49] build hierarchical consolidated process models from collections of event logs, which creates nested process fragments and allows to visualize the variability at different levels of abstraction. This does improve the maintainability problem, but still requires to specify all features of all variants. Declarative process specifications, on the other hand, do not require this knowledge to be modeled in advance. Therefore, we focus on declarative process specifications that allow to model different variants with varying degrees of variability.

Declarative process specifications offer change naturally through underspecification. Anything not specifically specified is subject to possible change. Numerous declarative approaches exist, of which most focus on supporting change during process enactment. Existing approaches consist of both formal and informal approaches. Informal approaches are those that lack either a formal representation of the used model, a formal specification, or both. Informal approaches include the work of Sadiq et al.

[50] which propose an algorithmic approach, and Pascalau et al. [51] which extend the compliance work in [52] which suffers from informal reduction rules. Other approaches offer change by specifying pre- and post-conditions for structured activities. Any change is allowed, as long as pre-conditions are met and post-conditions can be met. Such approaches include [53] and [54].

Existing formal approaches employ temporal logics to define the control flow of business processes during process enactment. Any change is allowed, as long as the temporal logic specifications are not violated. Approaches include [55–58]. Maggi et al. [59] extend upon [55] in order to support change pre-runtime, but report verification issues when encountering arbitrary cycles. Finally, Schunselaar et al. [60] extend upon [55] with configurable inclusion of activities and specifications.

De Giacomo et al. [61] extend the well-known imperative BPMN BPD specification with declarative flow control in order to develop a truly declarative business process specification. However, the approach does not consider parallel behavior or runtime consequences of the notation. For example, insertable tasks can be repeated any number of times, or can simply be avoided all together. In other words, tasks are either entirely optional, or required without any option for change. Having a required task which can be included in different places of the business process is simply

not possible, and considering parallel support would only increase these issues.

Marrella and Lespérance [62] present an approach to generate process templates by employing a partial-order planner. In this approach, the goal and cases have to be specified where the partial-order planner generates the possible paths that satisfy the constraints and properties of the domain. Our approach is orthogonal to the approach presented in [62], as we generate the constraints based on a set of input models.

Although declarative process specifications overcome the difficulties of imperative variability – which require knowledge of all change in advance – their abstract nature does introduce design difficulties which the intuitive imperative specification does not face. In this paper, we alleviate these issues through an automated approach that merges business process variants into a declarative specification of a process family which not only captures the specific variants, but also any variant in between those specified.

8. Conclusion

In this paper, we proposed an approach to automatically generate a declarative variability rule set from a set of business process variants. The method relies on the use of event structures and PES prefix unfoldings for the identification of behavior relations of the individual input process variants. These unfoldings are subsequently combined into a compound prime event structure, from which the different sets of variability rules are obtained, formulated using the well-known temporal logic CTL. The method allows to combine the rules defining the control-flow with rules specific to laws and regulations, such that each variant in a business process family is guaranteed to be compliant with those laws and regulations, while allowing the required flexibility for each different variant.

The proposed approach has been implemented in a tool called Apromore and subsequently evaluated on both a real life case from local eGovernment in the Netherlands and an existing set of models available online. We showed the applicability of the approach to newly identified process variants and demonstrated the ease of checking the conformance of that process variant to the obtained variability rule set.

In addition, we explored the effect of the complexity and size of the input variants on the resulting set of rules. The result show that the total amount of rules appears to be correlated with the amount of event labels. However, with an increasing number of input variants in a rule set, the total amount of rules tends to decrease, as the added variability (through the increase of different behavior allowed by the rules) generally reduces the “behavioral constraints” induced by the rules.

Finally, the evaluation showed that our approach for automatically generating a declarative set of variability rules is both effective and efficient. Even with a considerable increase in the complexity of the models, the entire variability rule set could be generated within 25 ms.

The automated assembly of rule sets with high variability, from either input processes, or runtime traces, has important implications for automated composition of variable service compositions. Similar to the assembly of these rule sets, rules for automated composition of service compositions can be created, with a high degree of variability, customizability, and personalization. In addition, runtime adaptations of such a composition can be detected and automatically incorporated in the original rule set. Finally, it can be used to assess the paths of a process variant that are not compatible with the generic variability rules.

For future work, we plan to apply this approach to a runtime setting, involving a variable service composition, in order to test its feasibility in a real-life business scenario.

Appendix A. CTL semantics

Kripke structures are used to interpret temporal logics. Temporal logics are formalisms that are able to reason over linear- or branching-time execution paths within system models. Linear-time temporal logics specify events over sequences of states known as paths. A path π is an infinite sequence of states $\pi = s_0, s_1, \dots$, such that the relation (s_i, s_{i+1}) exists for every $i \geq 0$. Branching-time temporal logics specify events over tree-like structures where branches represent possible execution paths. A Kripke structure is defined as follows [63]:

Definition 20 (Kripke Structure). Let AP be a set of atomic propositions. A Kripke structure K over AP is a quadruple $K = (S, S_0, R, L)$, 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 $(s, s') \in R$.
- $L : S \rightarrow 2^{AP}$ is a labeling function with the set of atomic propositions that are true in that state.

Computation Tree Logic (CTL) [16] is a branching-time temporal logic used with formal verification. CTL is especially useful when considering the different branching constructs available to compositions. The semantics of Computation Tree Logic (CTL) are defined on a Kripke structure M using the minimal set of CTL operators $\{\neg, \vee, EX, EG, EU\}$.

Definition 21 (Semantics of CTL). $M, s_i \models \phi$ means that the formula ϕ holds at state s_i of the model M . When the model M is understood, $s_i \models \phi$ is written instead. The relation $\models$ is defined inductively as follows:

$s_i \models \top$	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	$\exists (s_i, s_{i+1}) \in R \mid s_{i+1} \models \phi$
$s_i \models EG \phi$	iff	$\exists \pi = s_i, s_{i+1}, s_{i+2}, \dots \mid \forall n : (n \geq 0 \wedge s_{i+n} \models \phi)$
$s_i \models E[\phi U \phi']$	iff	$\exists \pi = s_i, s_{i+1}, s_{i+2}, \dots \mid \exists m : (m \geq 0 \wedge s_{i+m} \models \phi' \wedge \forall n : (0 \leq n < m : s_{i+n} \models \phi))$

Further CTL operators can be obtained through the following equivalences:

$$\begin{aligned}
 EF\phi &\equiv E[\text{true} U \phi] \\
 AF\phi &\equiv \neg EG\neg\phi \\
 AX\phi &\equiv \neg EX\neg\phi \\
 AG\phi &\equiv \neg EF\neg\phi \\
 A[\phi U \phi'] &\equiv \neg(E[\neg\phi' U \neg(\phi \vee \phi')] \vee EG\neg\phi')
 \end{aligned}$$

Appendix B. Transition graph

A transition graph is obtained from a Petri net through the following steps [15,1]. Consider that a transition t of a Petri net is enabled when $\forall p \in \bullet t : M(p) > 0$, then a multiset Y of transitions is enabled iff $\forall t \in Y, \forall p \in \bullet t : \sum_{(p,t) \in A} 1 \leq M(p)$, where 1 is the weight of the arc or number of tokens required per transition. The different multisets Y of parallel enabled transitions are defined through the following definition [15,1]:

Definition 22 (Parallel Enabled Binding Elements). The sets of all possible parallel enabled transitions $Y_{par}(M)$ at a marking M is obtained through the following steps:

- $Y(M) = \{t \mid \forall p \in \bullet t : M(p) > 0\}$ is the set of transitions enabled at a marking M .
- $Y_{sim}(M) = \{Y \mid Y \in \mathcal{P}(Y(M)) \wedge \forall t \in Y, \forall p \in \bullet t : \sum_{(p,t) \in A} 1 \leq M(p)\}$ are the sets of possible transitions that may fire simultaneously at marking M , with $\mathcal{P}(Y(M))$ being the power set of enabled transitions.
- $Y_{par}(M) = \{Y \mid Y \in Y_{sim}(M) \wedge \forall Y' \in Y_{sim}(M) : Y \not\subseteq Y' \wedge Y \neq \emptyset\}$ are the largest sets of parallel enabled transitions obtained from $Y_{sim}(M)$.

Using $Y_{par}(M)$, we convert a Petri net into a Kripke structure by creating states at each marking M_i for each set of transitions that can fire concurrently at a marking M_i , and then having each transition fire individually to find possible next states [15,1]:

Definition 23 (Transition Graph). Let AP be a set of atomic propositions. The transition graph of a Petri net with markings $M_0, \dots, M_n$ is a Kripke structure $K = (S, S_0, R, L)$ over AP , with:

- $AP = \{M_0, \dots, M_n\} \cup \{t \in Y \mid Y \in Y_{par}(M_0) \cup \dots \cup Y_{par}(M_n)\}$
- $S = \{s_i^Y \mid Y \in Y_{par}(M_i) : 0 \leq i \leq n\}$
- $S_0 = \{s_0^Y \mid Y \in Y_{par}(M_0)\}$
- $L(s_i^Y) = \{M_i\} \cup Y$
- $R = \{(s_i^Y, s_j) \mid t \in Y \wedge M_i \in L(s_i^Y) \wedge M_j \in L(s_j) \wedge M_i \xrightarrow{t} M_j \wedge L(s_i^Y) \setminus \{M_i, t\} \subseteq L(s_j)\} \cup \{(s_i^Y, s_i^Y) \mid Y = \emptyset\}$.¹¹

References

- [1] H. Groefsema, N.R.T.P. van Beest, M. Aiello, A formal model for compliance verification of service compositions, *IEEE Trans. Serv. Comput.* 11 (3) (2018) 466–479.
- [2] N.R.T.P. van Beest, P. Bulanov, J. Wortmann, A. Lazovik, Resolving business process interference via dynamic reconfiguration, in: 8th International Conference on Service Oriented Computing (ICSOC-2010), in: Lecture Notes in Computer Science, vol. 6470/2010, 2010, pp. 47–60.
- [3] N.R.T.P. Van Beest, E. Kaldeli, P. Bulanov, J.C. Wortmann, A. Lazovik, Automated runtime repair of business processes, *Inf. Syst.* 39 (2014) 45–79.
- [4] B.F. van Dongen, W.M. Van der Aalst, H.M. Verbeek, Verification of EPCs: Using reduction rules and petri nets, in: *CAiSE*, Vol. 3520, Springer, 2005, pp. 372–386.
- [5] R. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, *Inf. Softw. Technol.* 50 (12) (2008) 1281–1294.
- [6] M. La Rosa, H.A. Reijers, W.M.P. Van Der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, L. García-Bañuelos, APROMORE: An advanced process model repository, *Expert Syst. Appl.* 38 (6) (2011) 7029–7040.
- [7] L. García-Bañuelos, N.R.T.P. van Beest, M. Dumas, M. La Rosa, W. Mertens, Complete and interpretable conformance checking of business processes, *IEEE Trans. Softw. Eng.* 44 (3) (2018) 262–290.
- [8] C. Petri, Communication with Automata (Ph.D. thesis), Universität Hamburg, 1966.
- [9] W. Reisig, G. Rozenberg, Lectures on Petri Nets i: Basic Models: Advances in Petri Nets, Springer Science & Business Media, 1998.
- [10] J. Engelfriet, Branching processes of Petri nets, *Acta Inform.* 28 (1991) 575–591.
- [11] J. Esparza, Model checking using net unfoldings, *Sci. Comput. Program.* 23 (2–3) (1994) 151–195.
- [12] K.L. McMillan, D.K. Probst, A technique of state space search based on unfolding, *Form. Methods Syst. Des.* 6 (1) (1995) 45–65.
- [13] M. Nielsen, G. Plotkin, G. Winskel, Petri nets, event structures and domains, Part I, *Theoret. Comput. Sci.* 13 (1981) 85–108.
- [14] A. Armas-Cervantes, P. Baldan, M. Dumas, L. García-Bañuelos, Diagnosing behavioral differences between business process models: An approach based on event structures, *Inf. Syst.* 56 (2016) 304–325.
- [15] H. Groefsema, N.R.T.P. van Beest, Design-time Compliance of Service Compositions in Dynamic Service Environments, in: *Int. Conf. on Service Oriented Computing & Applications*, 2015, pp. 108–115.
- [16] E. Emerson, J. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, in: *Proc. of the 14th annual ACM symposium on Theory of computing*, 1982, pp. 169–180.
- [17] M.Y. Vardi, Branching vs. linear time: Final showdown, in: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2001, pp. 1–22.
- [18] H. Groefsema, N.R.T.P. van Beest, A. Armas-Cervantes, Automated compliance verification of business processes in Apromore, in: *Proceedings of the BPM Demo Track 2017*, 2017.
- [19] S. Maoz, J.O. Ringert, B. Rumpe, ADDiff: semantic differencing for activity diagrams, in: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ACM, 2011, pp. 179–189.
- [20] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan's unfolding algorithm, in: *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1996, pp. 87–106.
- [21] M.L. La Rosa, W.M. van der Aalst, M. Dumas, F.P. Milani, Business process variability modeling: A survey, *ACM Comput. Surv.* 50 (1) (2017) 2.
- [22] W. van der Aalst, S. Jablonski, Dealing with workflow change: identification of issues and solutions, *Comput. Syst. Sci. Eng.* 15 (5) (2000) 267–276.
- [23] M. Marin, R. Hull, R. Vaculín, Data centric bpm and the emerging case management standard: A short survey, in: *Business Process Management Workshops*, Springer, 2013, pp. 24–30.
- [24] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, W. van der Aalst, Process flexibility: A survey of contemporary approaches, in: *Advances in Enterprise Engineering I*, Springer Berlin Heidelberg, 2008, pp. 16–30.
- [25] M. Sinnema, S. Deelstra, P. Hoekstra, The COVAMOF derivation process, in: *Int. Conf. on Reuse of Off-the-shelf Components*, LNCS, Springer-Verlag, 2006, pp. 101–114.
- [26] W. van der Aalst, A. Dreiling, F. Gottschalk, M. Rosemann, M. Jansen-Vullers, Configurable process models as a basis for reference modeling, in: *Business Process Management*, Springer, 2005, pp. 512–518.
- [27] A. Schnieders, F. Puhmann, Variability mechanisms in E-business process families, in: *Proc. Int. Conf. on Business Information Systems*, Vol. 85, 2006, pp. 583–601.
- [28] S. Chang, S. Kim, A variability modeling method for adaptable services in service-oriented computing, in: *Int. Software Product Line Conference*, IEEE, 2007, pp. 261–268.
- [29] M. Rosemann, W. van der Aalst, A configurable reference modelling language, *Inf. Syst.* 32 (1) (2007) 1–23.
- [30] T. van Eijndhoven, M. Iacob, M. Ponisio, Achieving business process flexibility with business rules, in: *Enterprise Distributed Object Computing Conference*, IEEE, 2008, pp. 95–104.
- [31] F. Gottschalk, W. van der Aalst, M. Jansen-Vullers, M. La Rosa, Configurable workflow models, *Int. J. Coop. Inf. Syst.* 17 (2) (2008) 177–221.
- [32] F. Gottschalk, T. Wagemakers, M. Jansen-Vullers, W. van der Aalst, M. La Rosa, Configurable process models: Experiences from a municipality case study, in: *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, Springer, 2009, pp. 486–500.
- [33] M. Razavian, R. Khosravi, Modeling variability in business process models using UML, in: *Int. Conf. on Information Technology: New Generations*, IEEE, 2008, pp. 82–87.
- [34] C. Sun, M. Aiello, Towards variable service compositions using VxBPEL, in: *High Confidence Software Reuse in Large Systems*, 2008, pp. 257–261.
- [35] H. Hadaytullah, K. Koskimies, T. Systa, Using model customization for variability management in service compositions, in: *IEEE Int. Conf. on Web Services (ICWS 2009)*, 2009, pp. 687–694.
- [36] M. La Rosa, Managing Variability in Process-aware Information Systems, Vol. 25 (Ph.D. thesis), Queensland University of Technology Brisbane, Australia, 2009.
- [37] T. Nguyen, A. Colman, J. Han, Modeling and managing variability in process-based service compositions, in: *Int. Conf. Service-Oriented Computing (IC-SOC)*, Springer, 2011, pp. 404–420.
- [38] A. Hallerbach, T. Bauer, M. Reichert, Managing process variants in the process lifecycle, in: *Int. Conf. on Enterprise Information Systems (ICEIS'08)*, 2008, pp. 154–161.
- [39] W. van der Aalst, N. Lohmann, M. La Rosa, Ensuring correctness during process configuration via partner synthesis, *Inf. Syst.* 37 (6) (2012) 574–592.
- [40] W. van der Aalst, M. Dumas, F. Gottschalk, A. Ter Hofstede, M. La Rosa, J. Mendling, Preserving correctness during business process model configuration, *Form. Asp. Comput.* 22 (3–4) (2010) 459–482.
- [41] D. Schunselaar, E. Verbeek, W. van der Aalst, H. Raijers, Creating sound and reversible configurable process models using CoSeNets, in: *International Conference on Business Information Systems*, Springer, 2012, pp. 24–35.
- [42] D. Schunselaar, H. Verbeek, W. van der Aalst, H. Reijers, Petra: A tool for analysing a process family, in: *PNSE@ Petri Nets*, 2014, pp. 269–288.
- [43] M. La Rosa, M. Dumas, R. Uba, R. Dijkman, Merging business process models, in: *OTM Confederated International Conferences on the Move to Meaningful Internet Systems*, Springer, 2010, pp. 96–113.
- [44] M. La Rosa, M. Dumas, R. Uba, R. Dijkman, Business process model merging: An approach to business process consolidation, *ACM Trans. Softw. Eng. Methodol.* 22 (2) (2013) 11.

¹¹ Although Definition 23 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.

- [45] P. Bulanov, A. Lazovik, M. Aiello, Business process customization using process merging techniques, in: *Service-Oriented Computing (ICSOC)*, IEEE, 2011, pp. 1–4.
- [46] J. Buijs, B. van Dongen, W. van der Aalst, Mining configurable process models from collections of event logs, in: *Business Process Management*, Springer, 2013, pp. 33–48.
- [47] W. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, *Theoret. Comput. Sci.* 270 (1) (2002) 125–203.
- [48] F. Milani, M. Dumas, N. Ahmed, R. Matulevičius, Modelling families of business process variants: A decomposition driven method, *Inf. Syst.* 56 (2016) 55–72.
- [49] N. Assy, B.F. van Dongen, W.M. van der Aalst, Discovering hierarchical consolidated models from process families, in: *International Conference on Advanced Information Systems Engineering*, Springer, 2017, pp. 314–329.
- [50] S. Sadiq, M. Orłowska, W. Sadiq, Specification and validation of process constraints for flexible workflows, *Inf. Syst.* 30 (5) (2005) 349–378.
- [51] E. Pascalau, A. Awad, S. Sakr, M. Weske, Partial process models to manage business process variants, *Int. J. Bus. Process Integr. Manage.* 5 (3) (2011) 240–256.
- [52] A. Awad, G. Decker, M. Weske, Efficient compliance checking using BPMN-Q and temporal logic, in: *Business Process Management, BPM '08*, Springer, 2008, pp. 326–341.
- [53] I. Rychkova, G. Regev, A. Wegmann, Using declarative specifications in business process design, *Int. J. Comput. Sci. Appl.* 5 (3b) (2008) 45–68.
- [54] P. Dadam, M. Reichert, The ADEPT project: a decade of research and development for robust and flexible process support, *Comput. Sci. - R&D* 23 (2) (2009) 81–97.
- [55] M. Pesic, W. van der Aalst, A declarative approach for flexible business processes management, in: *Business Process Management Workshops*, Springer, 2006, pp. 169–180.
- [56] W. van der Aalst, M. Pesic, *DecSerFlow: Towards a Truly Declarative Service Flow Language*, Springer, 2006.
- [57] R. Demeyer, M. van Assche, L. Langevine, W. Vanhoof, Declarative workflows to efficiently manage flexible and advanced business processes, in: *International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ACM, 2010, pp. 209–218.
- [58] T. Hildebrandt, R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, *arXiv preprint arXiv:1110.4161*.
- [59] F. Maggi, M. Montali, M. Westergaard, W. van der Aalst, Monitoring business constraints with linear temporal logic: An approach based on colored automata, in: *Business Process Management*, Springer, 2011, pp. 132–147.
- [60] D. Schunselaar, F. Maggi, N. Sidorova, W. van der Aalst, Configurable declare: designing customisable flexible process models, in: *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, Springer, 2012, pp. 20–37.
- [61] G. De Giacomo, M. Dumas, F. Maggi, M. Montali, Declarative process modeling in BPMN, in: *Int. Conf. on Advanced Information Systems Engineering (CAISE)*, Springer, 2015, pp. 84–100.
- [62] A. Marrella, Y. Lespérance, A planning approach to the automated synthesis of template-based process models, *Serv. Oriented Comput. Appl.* 11 (4) (2017) 367–392.
- [63] E. Clarke, O. Grumberg, D. Peled, *Model Checking*, The MIT Press, Cambridge, Massachusetts and London, UK, 1999.