Supporting business process variability through declarative process families
Groefsema, H.; van Beest, N. R. T. P.

Published in:
Computers in Industry

DOI:
10.1016/j.compind.2024.104107

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:
2024

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

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.
Supporting business process variability through declarative process families

H. Groefsema a,∗, N.R.T.P. van Beest b

a Distributed Systems Group, Bernoulli Institute, University of Groningen, The Netherlands
b Software Systems Group, Data61 | CSIRO, Brisbane, Australia

A R T I C L E   I N F O

Keywords:
Business processes
Variability
Declarative
Process families
Temporal logic
Verification

A B S T R A C T

Organizations use business process management systems to automate processes that they use to perform tasks or interact with customers. However, several variants of the same business process may exist due to, e.g., mergers, customer-tailored services, diverse market segments, or distinct legislation across borders. As a result, reliable support for process variability has been identified as a necessity. In this article, we introduce the concept of declarative process families to support process variability and present a procedure to formally verify whether a business process model is part of a specified process family. The procedure allows to identify potential parts in the process that violate the process family. By introducing the concept of process families, we allow organizations to deviate from their prescribed processes using normal process model notation and automatically verify if such a deviation is allowed. To demonstrate the applicability of the approach, a simple example process is used that describes several variants of a car rental process which is required to adhere to several process families. Moreover, to support the proposed procedure, we present a tool that allows business processes, specified as Petri nets, to be verified against their declarative process families using the NuSMV2 model checker.

1. Introduction

When organizations perform their tasks and interact with customers, they execute a so-called business process. Business processes are collaborations between actors, which each fulfill roles to perform tasks that together achieve a value-added goal. In this regard, a business process consists of a collection of tasks performed in a specific order by actors fulfilling roles.

Business processes are subject to a large array of laws and regulations, and have to be continuously regulatory compliant to those laws and regulations. In order to aid the design of such business processes, a variety of modeling standards has been developed, including formalizations to visualize the behavior of business processes in the form of models. The most notable modeling standards are the Unified Modeling Language (UML) Activity Diagrams (OMG, 2015) and the Business Process Model and Notation (BPMN) (OMG, 2011), of which the latter is arguably the most popular in practice.

Business processes are typically automated using so-called business process management systems (BPMS). As a result of this automated support, business processes can be automatically checked against a set of specifications, or the actual behavior can be verified against the desired behavior as specified in the business process model.

Due to the operation of many modern organizations, different variants of the same business process may exist due to mergers, handling of different products, customer-tailored services, differences in market segments, or distinct legislation across borders, etc. (Aiello et al., 2010; La Rosa et al., 2017). The necessity of reliable support for process variability in the scope of legislation across local borders was already identified in 2009, when the SaS-LeG1 project was initiated by M. Aiello, P. Avgeriou, and J.C. Wortmann to model business process variants across local governments in the Netherlands, e.g., Aiello et al. (2010) and Groefsema et al. (2011). Since then, various core technologies have been developed using the initial outcomes of that project, in order to help business analysts to describe and analyze different process variants in such a way that duplication of specifications is avoided as much as possible, while retaining maintainability and compliance.

One approach towards process variability is so-called imperative variability, where the fixed parts of the process model are connected to specific regions that offer multiple execution alternatives, allowing the execution of multiple different process variants while using a single variable model. An alternative approach is to specify the process in a declarative manner, by constraining the allowed behavior in a process through a set of so-called variability rules. Under-specification of parts of a process (i.e. less-strict or even absence of rules to constrain that

* Corresponding author.
E-mail address: h.groefsema@rug.nl (H. Groefsema).

https://doi.org/10.1016/j.compind.2024.104107
Received 10 October 2023; Received in revised form 2 February 2024; Accepted 20 May 2024
Available online 28 May 2024
0166-3615/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
part of the process) then allows variable behavior that goes beyond a set of predefined alternatives. However, identifying whether such a specification holds for a certain process model is non-trivial.

In this article, we formally introduce the concept of process family as a specification consisting of a set of rules, and present a procedure for verifying whether a business process is part of a specified process family and for identifying potential parts in the process that violate the variability specification of that process family. To allow for such an application, we combine and build on methods from several earlier publications (van Beest et al., 2019; Groefsema et al., 2018, 2020), to provide a full framework of methods for verification and declarative specification of business processes.

Fig. 1 outlines the different concepts and their relations as defined in earlier approaches and the approach presented in this article. The verification of process models against formal requirements (1) has been presented in Groefsema et al. (2018, 2020). The automated conversion of (sets of) process models into a set of declarative rules (2) has been presented in van Beest et al. (2019). Together, these approaches allow for verification of declarative specifications against formal requirements (3). In this article, we formally define process families and verification of membership of a process model to a process family (3), formally distinguishing for the first time weak and strong process variants. The dotted area highlights the specific contribution of this article. When combining these individual approaches, this allows for the full spectrum of verification, also across process models and variants (4) and between formal requirements and declarative specifications of process families (5).

The remainder of the article is structured as follows. First, Section 2 provides an overview of the related work. Subsequently, Section 3 provides an overview of the preliminary concepts and definitions. Section 4 then describes the running example that will be used throughout the remainder of this article, before formally describing the method in Section 5. Next, the application of the method is highlighted followed by a performance evaluation in Section 6. Finally, the implications for industry of the presented method are summarized in Section 7.

2. Related work

Process variability is a useful feature to support the different process variants that may exist in modern organizations, due to, e.g., mergers, handling of different products, diverse market segments, or different regulatory requirements across countries. In the area of regulatory compliance and the verification of correctness of processes, the ability to adapt and change process instances is a desirable feature to ensure compliance in changing business circumstances. However, at the same time, each of the different process variants supported through variability are required to be compliant with policies and regulations. The clear connection between continuous correctness of processes and the ability to adapt processes on-the-fly was already identified in the SaS-LeG² project in 2009, leading to work on variability (Groefsema et al., 2011; Groefsema and Bucur, 2013) and runtime process adaptation to ensure correctness (van Beest et al., 2010, 2014).

Process variability is closely related to adaptability and flexibility, and are often used interchangeably. Adaptability offers change of a process at design-time, whereas flexibility allows deviations from the execution paths at runtime. On the other hand, variability deals with different versions of a process. Support of different versions of a process requires a certain amount of adaptability and/or flexibility management. In general, many approaches to variability can be distinguished, imperative and declarative variability (Aiello et al., 2010).

Imperative process variability introduces so-called variability points (Aiello et al., 2010; Sun and Aiello, 2008) in an imperative process model, where the fixed parts of the process are connected to specific regions that allow for multiple different options, such that multiple process variants can be executed while using a single variable model. Some of these approaches are integrated in popular process modeling languages (Delgado et al., 2020), such as BPMN, or even integrated in an automated Model-Driven Engineering approach (Calegari et al., 2020). In La Rosa et al. (2017), a full survey is presented on customizable business process models. However, imperative variability requires each variation to be specifically predefined as alternative options, which becomes a complex task as the number of possible variations increases.

An alternative approach is to specify the process in a declarative manner, by constraining the allowed behavior in a process through a set of so-called variability rules (van Beest et al., 2019) or process constraints (Pecic et al., 2007; van der Aalst et al., 2009; De Giacomo et al., 2015). Under-specification of parts of a process (i.e. less-strict or even absence of rules to constrain that part of the process) then allows variable behavior that goes beyond a set of predefined alternatives. Although declarative techniques for mining variability rules exist (see e.g. van Beest et al. (2019)), the resulting specification may comprise a large set of rules, rendering the identification of whether a certain process model is part of such a variability specification a non-trivial task.

A set of related process variants captured in a variability model or specification is commonly referred to as a process family (see e.g. Gröner et al. (2013) and Dimovski et al. (2015)). The importance of verifying whether a process variant is a member of a given process family has been acknowledged and an efficient reduction approach was developed by Dimovski et al. (2015). However, the ‘membership’ of a variant to a family depends on the degree of ‘relatedness’ that is desired. That is, a formal specification and distinction between strong and weak variants is essential for integrated verification of correctness of members against specifications and each other.

To the best of our knowledge, there exists no formal distinction between types of process variants in literature. Therefore, we formally introduce the concept of a process family and present a procedure for (i) verifying whether a business process model is part of a specified process family, and (ii) identifying potential parts in the process model that violate the variability specification of that process family.

3. Preliminaries

3.1. Petri nets with guards

Petri nets are a well-known formal modeling tool for business processes, and the use of Petri nets for the formalization of BPMN
models has been widely studied (see e.g. van Dongen et al. (2005), Dijkman et al. (2008) and La Rosa et al. (2011)). A Petri net can be formally defined as follows:

**Definition 1 (Petri Net).** A tuple \((P, T, A)\) is a 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

A Petri net execution state \(M : P \rightarrow \mathbb{N}_0\), also known as net marking, is a function that associates places with natural numbers (viz., place tokens). A marked net \(N = (P, T, A, M_0)\) is a Petri net \((P, T, A)\) with an initial marking \(M_0\). Additionally, we introduce a labeling function \(\lambda : P \cup T \rightarrow \mathcal{L}\), which assigns a label to a place or transition.

Places and transitions are referred to as nodes. The preset of a node is denoted by \(M\). Whether the preset of that transition contains enough tokens). A guard \(\lambda\) an an condition is represented as a guard, which is associated to a transition defined over a transition \(M\). Values needs to be assigned to the variables in the respective guard such that \(\lambda\) is generated by the following grammar, assuming \(p \in AP\): \(\phi ::= \top \mid \bot \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid A\phi \mid \exists\phi \mid \forall\phi\).

Often used with formal verification, CTL is especially useful when considering the different branching constructs.

**Definition 4 (CTL Syntax).** The language of well-formed CTL formulas is generated by the following grammar, assuming \(p \in AP\): \(\phi ::= \top \mid \bot \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid A\phi \mid \exists\phi \mid \forall\phi\).

**Definition 5 (Semantics of CTL).** \(M, i, s \vdash \phi\) means that the formula \(\phi\) holds at state \(s\) of the model \(M\). When the model \(M\) is understood, \(s, t \vdash \phi\) is written instead. The relation \(\vdash\) is defined inductively as follows:

3.2. Model checking

One of the most successful and well-known approaches towards formal verification is model checking. Model checking is a technique that automatically and systematically explores a system model to formally check whether a given formal specification holds at each explored state.

Such system models are typically represented by labeled transition systems (LTS). A Kripke structure is a specific variation of an LTS with a labeling function over its nodes, which is often used to interpret a formal specification formulated using temporal logics. Kripke structures are defined as follows Clarke et al. (1999).
terminates after canceling the service. If the credit card transaction is successful, the key is exchanged after which the process is finished.

Another variant of the car rental process is illustrated in Fig. 3, where the license is checked directly after selection of the service and the extras are only added after selecting the car. The license is copied before the insurance agreement, instead of concurrently.

Finally, an additional variant of the car rental process can be considered, as shown graphically in Fig. 4, in which a more premium car rental service is offered where all extras are included as standard and an upgraded vehicle is provided. However, in order to be eligible for this service, the customer is required to be at least 25 years old, which is checked right after “Check license”.

Together, these process models represent a set of process variants (cf. Fig. 1) implementing the car rental process, which is usually provided through a number of cloud-based services, exposed by several different service providers. To use these services, the user must comply with a number of rules as stated by their service providers, as well as with a number of business rules and regulations provided by government. These rules form three distinct declarative process families; (i) a governmental regulatory family specifying the government regulations each rental company must comply with, (ii) an insurance statutory family specifying the insurance statutes for car rental, and (iii) a franchise procedural family indicating specific rules for each franchise owner.

The rules for the process families are as follows:

1. Governmental regulatory family:
   (a) The driver license must always be checked before it is being copied.
   \[ \neg E \neg ("Check license" \cup "Copy license") \]
   (b) The driver license must always be copied before keys are exchanged.
   \[ \neg E \neg ("Copy license" \cup "Exchange key") \]
   (c) When the license check fails, keys are never exchanged.
   \[ k_\neg ("License_ok = false" \Rightarrow \neg EF "Exchange key") \]

2. Insurance statutory family:
   (a) The activity "Copy license" should always occur before "Create insurance agreement".
   \[ \neg E \neg ("Copy license" \cup "Create insurance agreement") \]
(b) The activity “Create insurance agreement” should never occur concurrently with “Copy license”.
\[\neg \EF (“Create insurance agreement” \land “Copy license”)\]

3. Franchise procedural family:
(a) After a car is selected, the activity “Charge card” is always eventually performed.
\[\AG (\text{Select car} ) \Rightarrow \EF (\text{Charge card})\]
(b) After a car is selected either the activity “Exchange key” or “Cancel service” is eventually executed.
\[\AG (\text{Select car} ) \Rightarrow \EF (\text{Exchange key} \lor \text{Cancel service})\]

Throughout the description of our method in this article, we will use a single rule (2b) to show how a business process can be checked on rules within a process family.

5. Method

Declarative variability can be defined as an LTS adhering to a set of specifications that represent a process family. In this way, the membership of an LTS to a process family can be formally verified using model checking techniques. Of course, sets of specifications cannot be directly verified on business process models or their marked net representations. As a result, an LTS must be obtained from the marked net before formal verification is applied. This LTS then allows the interpretation of CTL formulas on the possible executions of the marked net. Subsequently, when such a verifiable model, obtained from the marked net of a business process, is proven to adhere to the set of specifications that represent a process family, that business process model is said to belong to the process family. In this section, the above concepts are formally defined and explained in detail.

5.1. Declarative process families

Declarative process families are defined by adherence to a set of specifications. Such a set of specifications consists of a number of temporal logic formulas over a set of atomic propositions that represent labels of activities in business process models. A process family is defined as follows:

**Definition 6 (Process Family).** Let \(AP\) be a set of atomic propositions describing labels of process elements. A process family \(F\) over \(AP\) is a set of temporal logic formulas using \(p \in AP\).

For example, consider the process family \(F_{nc}\) below, which defines a family of processes where the “Create insurance agreement” activity may not be executed concurrently with the “Copy license” activity. Given the running example of Fig. 2, it is easy to see that this is not the case for that process model.

\[F_{nc} = \{\neg \EF (“Create insurance agreement” \Rightarrow “Copy license”)\} \cup \{“Create insurance agreement”, “Copy license”\} \subseteq AP_{nc}\]

A process family, a variant is an LTS, i.e., a Kripke structure (Definition 3), that adheres to the process family specification. Two types of variants are defined: weak and strong variants. A variant is considered strong if it includes all propositions used in the process family specification, and weak if it contains some propositions used in the process family specification. Note that variants allow any additional propositions to be used within the LTS as long as it adheres to the process family specification. Weak and strong variants are defined formally as follows:

**Definition 7 (Weak Variant).** Let \(F\) be a process family over the set of atomic propositions \(AP\). A Kripke structure \(K = (S, S_0, R, L)\) over \(AP\) is a weak variant if \(AP \cap AP' \neq \emptyset\) and \(\forall \phi \in F, s_0 \in S_0 : K, s_0 \models \phi\).

**Definition 8 (Strong Variant).** Let \(F\) be a process family over the set of atomic propositions \(AP\). A Kripke structure \(K = (S, S_0, R, L)\) over \(AP\) is a strong variant iff \(AP \subseteq AP'\) and \(\forall \phi \in F, s_0 \in S_0 : K, s_0 \models \phi\).

Assume two variants of the process family \(F\) over \(AP\) exist: \(K^x\) and \(K^y\) over \(AP^x\) and \(AP^y\) respectively. If weak variants, \(K^x\) and \(K^y\) might be completely disjointed because of the weak requirements that \(AP \cap AP^x \neq \emptyset\) and \(AP \cap AP^y \neq \emptyset\). That is, it is possible that \(AP^x \cap AP^y = \emptyset\). Similarly, if strong variants, the atomic propositions of each variant that are not in \(AP\) need not overlap. That is, it is possible that \(AP^x \cap AP \neq \emptyset\) \(\land\) \((AP^y \cap AP) = \emptyset\). Note, however, that a process family \(F\) is free to specify overlap of atomic propositions used in variants by including formulas that require inclusion of certain atomic propositions, e.g., requiring \(t \in AP\) by including the rule \(\EF t\).

In this way, process families can require their variants to always be strong.

5.2. Obtaining a verifiable model

To support full expressiveness of concurrent executions, a special type of LTS will be used for verification, which is referred to as a transition graph (Groefsema et al., 2018). Transition graphs are Kripke structures obtained from marked nets that are able to capture parallel behavior, such that consecutive transitions on parallel branches can be identified individually.

To obtain a transition graph from a marked net, a set of states (including the initial states), a set of relations between states, and a set of atomic propositions to label states must be created. To create states, the different sets of transitions that can be enabled simultaneously at a marking \(M\) must be obtained first. Consider that a transition \(t \in T\) of a Petri net is enabled when \(\forall p \in \mathcal{S} : M(p) > 0\) holds, then the set of all enabled transitions given a marking \(M\) can be defined as the set:

\[\mathcal{Y}(M) = \{ t \mid t \in T, \forall p \in \mathcal{S} : M(p) > 0\}\]

Consider, for example, the excerpt of the car rental process depicted in Fig. 5. In this example, the marked net has advanced in such a way that it contains a token at \(p_4\), i.e., its marking is \(M_1 = 1p_4\). Since \(p_4\) contains a token, both \(s_2\) and \(s_3\) are enabled and may fire if their guards evaluate true. In other words, the set of enabled transitions contains both \(s_2\) and \(s_3\);

\[\mathcal{Y}(M_1) = \{ s_2, s_3 \}\]

Suppose \(s_2\) fires such that \(M_2 = \frac{s_2}{s_3} M_1\), resulting in the marking \(M_2 = 1p_4 + 1s_2\). In this case, the set of enabled transitions for \(M_2\) equals \(\mathcal{Y}(M_2) = \{ t_4, t_5 \}\).

Note that the sets of enabled transitions only contain those transitions that are \(\mathcal{I}\) enabled individually, and do not determine whether they can fire simultaneously. Given marking \(M_2\) in the example above, for instance, \(s_2\) and \(s_3\) are both enabled but cannot fire both. That is, they are not \(\mathcal{I}\) enabled in parallel. In case of marking \(M_3\) on the other hand, \(t_4\) and \(t_5\) are enabled in parallel, as both of them can fire.

Given a marking \(M\) and an interpretation \(I\) providing a set of values for the variables of the guards \(X \subseteq X\), a set of transitions \(Y\) is enabled in parallel if the following Boolean expression holds:

\[E(M, I, Y) \overset{\text{def}}{=} (I \not\exists_{i \in T} G_i) \land (\forall t \in Y, \forall p \in \mathcal{S} : M(p) \geq [p \cdot \cap Y])\]
Using the power set of the enabled transitions \( P(\mathcal{Y}(M)) \) and the function \( E(M, I, Y) \) to determine whether a set is enabled in parallel, all the different sets of parallel enabled transitions at marking \( M \) are obtained as the set:

\[
\mathcal{Y}_p(M) = \{ Y \mid Y \in P(\mathcal{Y}(M)) \} ∩ (\exists I : E(M, I, Y) ∧ (∃ t ∈ \mathcal{Y}(M) : E(M, I, Y ∪ \{ t \} ) ) )
\]

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

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

In Fig. 5, even though both \( s_2 \) and \( s_3 \) are enabled for the marking \( M_i = p_{s_2} \), the place \( p_t \) only contains a single token while the guards \( G_{s_2} = \neg License\_ok \) and \( G_{s_3} = License\_ok \) contradict. As a result, only \( t \) enabled transition is fired individually to find possible next markings. Only when one of the enabled transitions may actually fire. This fact is reflected over transition ids as labels, sometimes it is easier to use additional labels over the states of the transition graph. For example, the labels of transitions could be included in the set of atomic propositions and using \( \lambda(t) \) if \( t \in Y \). Similarly, transition guards could be included using \( G_t \in L(\phi) \) if \( t \in Y \). That is, the state labeled with \( s_4 \) in Fig. 7 could also be labeled with the guard of transition \( s_3 \) (i.e., \( G_{s_3} = \neg License\_ok \)), which could subsequently be used to verify for the possible outcome where the license was checked and somehow not accepted.

5.3. Process family membership

Now that verifiable models can be obtained through transition graphs, membership of process families can be verified using model checking techniques. Given a process family \( F \) over \( AP \), a transition graph \( K = (S, S_0, R, L) \) over \( AP \) is a variant iff \( \forall \phi \in F, s_0 \in S_0 \) : \( K, s_0 \vDash \phi \), and \( AP \cap AP' \neq \emptyset \) (weak variant, Definition 7) or \( AP \subseteq AP' \) (strong variant, Definition 8). As a consequence, a marked net is a process variant of the process family \( F \) if the transition graph obtained from that marked net (Definition 9) is a variant. More formally:

**Definition 10 (Process Variant).** Let \( AP' \) be a set of atomic propositions representing markings and transitions of a marked net \( N = (P, T, A, M_0) \), and \( F \) be a process family over a set of atomic propositions \( AP \). The marked net \( N \) is a weak/strong process variant of the process family \( F \) iff the transition graph \( K = (S, S_0, R, L) \) over \( AP' \), and obtained from \( N \), is a weak/strong variant.

As an extension, a business process model is a process variant of a process family if the marked net it is represented by is a member of the process family. A business process model, marked net, or LTS is a member of a process family if it is a (process) variant. For example, consider a process family that requires that the “Create insurance agreement” and “Copy license” activities are never performed concurrently in the car rental example. This requirement can be specified as the process family \( F_p = \{ \neg\lambda(\text{"Create insurance agreement" ⇒ } \lambda(\text{"Copy license"})) \} \), or – using the transition ids of the marked net – as \( F_p' = \{ \neg\lambda(t_2 \Rightarrow t_4) \} \). This formula specifies that there may not exist a path that eventually contains a state that is labeled with \( t_4 \) if it is also labeled with \( t_2 \). The result of this formula is depicted graphically using the transition graph of the car rental process in Fig. 8. As illustrated by the highlighted path, there exists a path to a state that is labeled with both the “Create insurance agreement” activity (i.e., \( t_2 \)) and the “Copy license” activity (i.e., \( t_4 \)). As a result, the transition graph is not a variant of \( F_p' \) and variant \( N_3 \) of the car rental process (Fig. 2) is not a member of the process family \( F_p' \). On the other hand, it is easy to see that variant \( N_2 \) of the car rental process (Fig. 3) is a member of the process family \( F_r \).

6. Evaluation

The combined techniques described throughout this article have been extensively evaluated across several earlier publications with respect to multiple characteristics (van Beest et al., 2019; Groefsema et al., 2018, 2020; Groefsema and van Beest, 2015). In this article, we have rerun those experiments with the additionally presented features and summarize their outcomes and implications. We first describe the tool supporting the combined techniques presented here, followed by an evaluation on applicability, performance, and complexity.

6.1. Tool support

The techniques described in this section are supported by two open source Java packages; BPMPetriNet³ and BPMVerification.⁴

---

The BPMVerification package provides verification support for the Petri net implementation of BPMPetriNet. It offers support for the techniques described throughout this article, as well as the conditional Petri net implementation described in van Beest et al. (2019). A command line entry point class is included in the BPMVerification package. To verify regulatory compliance of a business process model, the class takes as input a place-transition net and a specification in the form of an XML file, translates the place-transition net to a transition graph, reduces the state space, verifies it against the provided specification, and returns the result. To perform the actual model checking of the transition graph, the package relies on the NuSMV2\textsuperscript{6} or NuXMV\textsuperscript{7} model checker. The class is called with the following options:

```
java CommandlineVerifier
-c,-checker <arg> model checker binary location
-l,-log <arg> the log level, either critical, error, warning, info (default), verbose, or debug
-n,-net <arg> type of Petri net to use, either ptnet (default) or dnet
-o,-output <arg> output directory path
-p,-pnml <arg> pnml file path
-s,-spec <arg> specification file path
-v,-verifier <arg> type of verifier to use, either kripke, stutter, or multi (default)
```

### 6.2. Applicability

The applicability of the approach was evaluated in van Beest et al. (2019), Groefsema et al. (2018, 2020), and Groefsema and N.R.T.P. van Beest (2020), the verification approach was evaluated on a real regulatory compliance case-study from the telecom sector in Australia. Telecommunications service providers in Australia are required to comply to the Telecommunications Consumer Protections (TCP) code of conduct, which is registered by the Australian Communications and Media Authority. As such, the applicability was evaluated by verifying a number of TCP regulations on a real customer support process.

In van Beest et al. (2019), the variability specification (i.e., process family) was evaluated using a real case-study from the Dutch government. In the Netherlands, the Wet Maatschappelijke Ondersteuning (Social Support Act, 2006) is a law that is defined nationally yet executed locally at the municipality level. As a result, the execution of the law is heavily affected by local needs. This, in turn, results in different variants of the same process for each of the municipalities that exist in the Netherlands. Using this case study, we demonstrated how to encode process families along different facets and degrees of strictness. Moreover, we showed how such a process family can be obtained automatically.

The running example described in Section 4 provides different sets of rules, each representing a process family to which three distinct variants of the car rental process may belong. Table 1 lists the results of checking the process family memberships for the three processes of the car rental service. When observing Table 1, one can conclude that only process $N_3$ is a variant of all three process families. Process $N_2$ is a variant of the governmental and franchise process families, but not a variant of the insurance process family because it violates rule 2(b) (as shown in Fig. 8). Similarly, process $N_1$ is a variant of the governmental and insurance process families, but not a variant of the franchise process family because it violates rule 3(a), which states that “Charge card” is always eventually performed after “Select car”. When considering Fig. 4, it is easy to see that the activity “Charge card” is actually performed before “Select car”.

Table 2 provides an overview of the time required to convert each of the three models against the respective process families. All conversions require less than 3 ms. The reduction of each respective model against the process families takes around 0.5 ms. The partitioning of the model

---

\textsuperscript{5} https://pnml.org.

\textsuperscript{6} https://nusmv.fbk.eu/.

\textsuperscript{7} https://nuxmv.fbk.eu/.

---

![Fig. 7. Transition graph derived from Fig. 2.](image_url)

![Fig. 8. Graphical depiction of the counter example found on the Kripke structure of the car rental example for the CTL formula $\neg\Box(t_4 \Rightarrow t_5)$.](image_url)

---

Table 1

<table>
<thead>
<tr>
<th>Process family</th>
<th>Process $N_1$</th>
<th>Process $N_2$</th>
<th>Process $N_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Government</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2. Insurance</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3. Franchise</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Government</th>
<th>Insurance</th>
<th>Franchise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>0.5 ms</td>
<td>0.5 ms</td>
<td>0.5 ms</td>
</tr>
<tr>
<td>2nd</td>
<td>3 ms</td>
<td>3 ms</td>
<td>3 ms</td>
</tr>
</tbody>
</table>
is performed during the conversion itself, while the subsequent reductions are computed in parallel per process family. An overview of the performance of the reduction and the resulting complexity properties (amount of states (S), amount of relations (R) and amount of atomic propositions (AP)) is provided in Table 3.

6.3. Correctness

With respect to the correctness of the verification approach, in Groefsema et al. (2018) and Groefsema and van Beest (2015) we have shown that CTL can be interpreted on the possible executions of a Petri net by converting the Petri net into a transition graph (Definition 9) and interpreting a set of newly defined CTL semantics on the reachability graph of that Petri net. A reachability graph captures the possible executions of a Petri net. Since it is possible to also obtain this reachability graph from the transition graph, it is possible to define the semantics of CTL on the reachability graph as it were interpreted on the transition graph. Using these semantics, we can interpret CTL on the possible executions of a Petri net. Although these semantics were defined using the reachability graph of a colored Petri net, and were limited to CTL minus the next operator (CTL-X), it is easy to define the same translations and semantics for the more straightforward Petri nets with guards and the full set of CTL operators — or even the set of CTL* operators.

6.4. Expressive power

The expressive power of the approach has been evaluated in Groefsema et al. (2018). Results show that the transition graph is uniquely able to capture parallel firing of transitions as well as local next transition firings, i.e., the next transition to fire on a branch, including those that are interleaved due to parallel firings. This is especially useful, as it provides the ability to verify next executions with the same CTL formula on both interleaved and non-interleaved branches.

6.5. State space

The size of the state space of the obtained transition graph has been evaluated for both the full graphs (Groefsema et al., 2018; Groefsema and van Beest, 2015), as well as partial graphs based on guard conditions (Groefsema et al., 2020). Results confirm that, due to a lack of complete linearization of parallel executing transitions and the ability to correctly join converged paths, the state space features a diminished growth. In addition, we have shown that the obtained transition graphs can be significantly reduced (without losing the required expressive power) by using CTL-X, the subsequent removal of irrelevant atomic propositions, and the calculation of a stutter equivalent transition graph. Moreover, it is possible to split formulas into multiple smaller sets, each resulting in a much smaller reduced transition graph upon which the related specification set can be verified, which results in a significant performance gain. As such, the size of the reduced transition graph is directly related to the number of atomic propositions used within the set of formulas.

6.6. Performance and scalability

The performance of the algorithm that obtains a transition graph from a Petri net has been evaluated in Groefsema et al. (2018) and Groefsema and van Beest (2015). Results of the performed experiments confirm that this conversion algorithm performs well, and allows us to obtain transition graphs from Petri nets, including those with large parallel regions, within a feasible time.

However, the tool set presented in this article supports full declarative variability and process families, incorporating a number of extensions with respect to supported features and expressivity, such as support for conditions and guards, inferences of atomic propositions, etc. As such, we have conducted a performance and scalability test on a set of synthetic process models, featuring an increasing complexity with respect to states, relations and atomic propositions.

The evaluation was performed on an Apple M1 Max chip with 32 GB, running JVM 17 with 2 GB allocated memory. To eliminate load times, we executed each test ten times and recorded the average times of eight executions, removing the fastest and the slowest executions.

The synthetic process models each contain a structured XOR-block or AND-block, where the complexity is generated through an increasing amount of branches (n) and increasing length of each branch (m). Table 4 provides an overview of the different models and their respective conversion performance, reduction and resulting complexity properties (amount of states (S), amount of relations (R) and amount of atomic propositions (AP)). The models were generated and subsequently reduced using three generically applicable rules, requiring a causality relation between a pair of transitions:

1. “\text{node\_start}” is always eventually followed by “\text{node\_end}”.
\[ \Box \langle \text{node\_start} \rangle \Rightarrow \Box \langle \text{node\_end} \rangle \]
2. “\text{node3}” is always eventually followed by “\text{node\_end}”.
\[ \Box \langle \text{node3} \rangle \Rightarrow \Box \langle \text{node\_end} \rangle \]
3. “\text{node7}” is always eventually followed by “\text{node\_end}”.
\[ \Box \langle \text{node7} \rangle \Rightarrow \Box \langle \text{node\_end} \rangle \]

As shown in Table 4, the amount of possible executions (and, consequently, the complexity of the state model) is limited for models composed of an XOR-block, as it depends on the amount of branches (n) of the model. As such, the conversion is very fast (max. <7 ms), even for very large models (e.g. XOR 10 × 10). However, the complexity of the models composed of an AND-block is combinatorial with respect to the amount of branches (m) and length of each branch (n). As a result, those models are much more sensitive to an increasing size, which is clearly reflected in their increasing required conversion time. Nevertheless, the results show that even for more complex models (such as e.g. AND 4 × 8 or AND 5 × 5) the conversion itself requires less than 2.5 s, with a subsequent reduction using the three rules taking around 10 ms. Consequently, the significant increase in expressivity and features compared to Groefsema et al. (2018) only comes at a small cost in performance, as unrealistically complex models (a 4 × 8 concurrent structure results in 9.96 · 10^10 different executions due to interleaving) still perform in an acceptable amount of time.

7. Conclusion

Different variants of business processes are an innate result of modern international and customer tailored businesses. To support such

<table>
<thead>
<tr>
<th>Model</th>
<th>Family</th>
<th>Reduction (ms)</th>
<th>S</th>
<th>R</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>Governmental regulatory family</td>
<td>0.57</td>
<td>10</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Insurance statutory family</td>
<td>0.57</td>
<td>11</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Franchise procedural family</td>
<td>10</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>N2</td>
<td>Governmental regulatory family</td>
<td>0.47</td>
<td>11</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Insurance statutory family</td>
<td>0.47</td>
<td>8</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Franchise procedural family</td>
<td>10</td>
<td>12</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>N3</td>
<td>Governmental regulatory family</td>
<td>0.53</td>
<td>10</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Insurance statutory family</td>
<td>0.53</td>
<td>8</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Franchise procedural family</td>
<td>9</td>
<td>11</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Reduction for the specific process families.

Table 2 Performance test of the process families in the running example.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time (ms)</th>
<th>S</th>
<th>R</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>2.85</td>
<td>17</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>N2</td>
<td>2.25</td>
<td>14</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>N3</td>
<td>2.16</td>
<td>15</td>
<td>18</td>
<td>24</td>
</tr>
</tbody>
</table>
business process variants, we built on several earlier publications to introduce the concept of declarative process families and defined a procedure to formally verify whether a business process is a variant and, therefore, part of such a process family. Two types of variants are defined: weak variants that adhere to all formulas and include some atomic propositions (i.e., labels of transitions) used in the process family, and strong variants that include all such atomic propositions.

We demonstrated the applicability of the procedure by applying the proposed approach on a simple running example that describes several variants of a car rental process, which is required to adhere to several process families. Moreover, to support the procedure, we presented a tool that allows a Petri net with guards to be verified against a declarative process family using the NuSMV2 model checker.

Evaluations have shown that the proposed procedure is applicable on real-life processes, such as customer support and governmental processes. The procedure allows for correct reasoning over the possible executions of such processes. Moreover, the procedure is highly expressive as it is uniquely able to capture parallel executions as well as local next executions. The state space, scalability, and performance evaluations show that the procedure performs well, and allows for the verification of business processes, including those with large parallel regions, within a feasible time.

CRediT authorship contribution statement

H. Groefsema: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing.
N.R.T.P. van Beest: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References


Table 4

<table>
<thead>
<tr>
<th>Model (m × n)</th>
<th>Time (ms)</th>
<th>S</th>
<th>R</th>
<th>AP</th>
<th>Reduction (ms)</th>
<th>S</th>
<th>R</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR 2 × 2</td>
<td>0.72</td>
<td>7</td>
<td>8</td>
<td>0.05</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 2 × 5</td>
<td>0.92</td>
<td>13</td>
<td>14</td>
<td>0.05</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 2 × 10</td>
<td>1.22</td>
<td>23</td>
<td>24</td>
<td>0.06</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 3 × 2</td>
<td>0.85</td>
<td>9</td>
<td>11</td>
<td>0.05</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 3 × 5</td>
<td>1.07</td>
<td>18</td>
<td>20</td>
<td>0.05</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 3 × 10</td>
<td>1.54</td>
<td>33</td>
<td>35</td>
<td>0.07</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 4 × 2</td>
<td>0.93</td>
<td>11</td>
<td>14</td>
<td>0.04</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 4 × 5</td>
<td>1.49</td>
<td>23</td>
<td>26</td>
<td>0.06</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 4 × 10</td>
<td>1.82</td>
<td>43</td>
<td>46</td>
<td>0.08</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 5 × 2</td>
<td>1.05</td>
<td>13</td>
<td>17</td>
<td>0.05</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 5 × 5</td>
<td>1.75</td>
<td>28</td>
<td>32</td>
<td>0.06</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 5 × 10</td>
<td>2.13</td>
<td>53</td>
<td>57</td>
<td>0.08</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>XOR 10 × 10</td>
<td>6.61</td>
<td>103</td>
<td>112</td>
<td>0.11</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>AND 1 × 2</td>
<td>0.53</td>
<td>5</td>
<td>5</td>
<td>0.04</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>AND 2 × 2</td>
<td>0.98</td>
<td>11</td>
<td>15</td>
<td>0.04</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>AND 2 × 5</td>
<td>2.2</td>
<td>38</td>
<td>63</td>
<td>0.07</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>AND 2 × 10</td>
<td>6.05</td>
<td>123</td>
<td>223</td>
<td>24</td>
<td>0.15</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 3 × 2</td>
<td>2.3</td>
<td>29</td>
<td>57</td>
<td>0.07</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>AND 3 × 5</td>
<td>21.47</td>
<td>218</td>
<td>543</td>
<td>19</td>
<td>0.27</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 3 × 10</td>
<td>137.71</td>
<td>1333</td>
<td>3633</td>
<td>34</td>
<td>1.33</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 4 × 2</td>
<td>9.45</td>
<td>83</td>
<td>219</td>
<td>12</td>
<td>0.13</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 4 × 5</td>
<td>208.86</td>
<td>1298</td>
<td>4323</td>
<td>24</td>
<td>1.44</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 4 × 8</td>
<td>1182.29</td>
<td>6563</td>
<td>2331</td>
<td>36</td>
<td>7.86</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 5 × 5</td>
<td>2234.12</td>
<td>7778</td>
<td>32403</td>
<td>29</td>
<td>10.84</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 5 × 10</td>
<td>1755.29</td>
<td>4098</td>
<td>18435</td>
<td>22</td>
<td>5.28</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>AND 7 × 2</td>
<td>1225.62</td>
<td>2189</td>
<td>10209</td>
<td>18</td>
<td>2.89</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

No data was used for the research described in the article.