Software and Systems Modeling (2023) 22:667–685
https://doi.org/10.1007/s10270-022-01069-y

SPECIAL SECTION PAPER

Automaton-based comparison of Declare process models

Nicolai Schützenmeier[1] · Martin Käppel[1] · Lars Ackermann[1] · Stefan Jablonski[1] · Sebastian Petter[1]

Received: 1 March 2022 / Revised: 26 October 2022 / Accepted: 14 November 2022 / Published online: 5 December 2022

Abstract
The Declare process modeling language has been established within the research community for modeling so-called flexible processes. Declare follows the declarative modeling paradigm and therefore guarantees flexible process execution. For several reasons, declarative process models turned out to be hard to read and comprehend. Thus, it is also hard to decide whether two process models are equal with respect to their semantic meaning, whether one model is completely contained in another one or how far two models overlap. In this paper, we follow an automaton-based approach by transforming Declare process models into finite state automatons and applying automata theory for solving this issue.

Keywords Business process modeling · Declare · Model comparison · Declarative process management · Automata theory

1 Introduction

In business process management (BPM), two opposing classes of business processes can be identified: routine processes and flexible processes (also called knowledge-intensive, decision-intensive or declarative processes) [1, 2]. For the latter, in the last years a couple of different process modeling languages such as Declare [3], Multi-Perspective-Declare (MP-Declare) [4], DCR graphs [5] and the Declarative Process Intermediate Language (DPIL) [6,7] emerged. These languages are called *declarative modeling languages*. They describe a process by restrictions (so-called *constraints*) over the behavior, which must be satisfied throughout process execution. Especially Declare has become a widespread and frequently used modeling language in the research area of modeling single-perspective (i.e., focusing on the control flow) and flexible processes.

This paradigm guarantees more flexibility than the *imperative* one, which is the modeling standard for routine processes. But on the other hand it turned out that declarative process models are for several reasons hard to read and understand, which affects the execution, modeling and maintenance of declarative process models in a negative way: the large degree of flexibility offers the modeler a multitude of options to express the same fact. Hence, the same process can be described by very different declarative process models (cf. Sect. 2). In general, declarative process models possess a high risk for over- or underspecification, i.e., the process model forbids valid process executions or allows process executions that do not correspond to reality, respectively. Such a wrong specification is often caused by hidden dependencies [8], i.e., implicit dependencies between activities that are not explicitly modeled but occur through the interaction of other dependencies. The Declare modeling language relies on linear temporal logic (LTL) [3]. Hence, constraints and process models, respectively, are represented as LTL formulas. Although there is a set of common Declare templates, this set is not exhaustive in the sense that sometimes plain LTL formulas are necessary to complete a process specification. Also for defining customized templates for reuse (i.e., if a dependency between more than two activities should be expressed) modelers are not aware of working with plain LTL. This deficiency increases since a canonical standard form for LTL formulas does not exist, so in general, these formulas are not

Communicated by E. Serral Asensio, J. Stirna, J. Ralyté, and J. Grabis.

✉ Nicolai Schützenmeier
nicolai.schuetzenmeier@uni-bayreuth.de

Martin Käppel
martin.kaeppel@uni-bayreuth.de

Lars Ackermann
lars.ackermann@uni-bayreuth.de

Stefan Jablonski
stefan.jablonski@uni-bayreuth.de

Sebastian Petter
sebastian.petter@uni-bayreuth.de

1 University of Bayreuth, Bayreuth, Germany

unique. Enriching the predefined constraints with plain LTL exacerbates the problem of understanding such models.

Therefore, there is a high interest to keep a process model as simple as possible without deteriorating conformance with reality. However, changing or simplifying such a process model bears the risks described above, i.e., over- and under-specification. Hence, model checking, especially comparing models on equality, becomes an important task for modeling and verifying declarative process models. Most of the time this is achieved by simulating process executions of different lengths (so-called trace length) and checking their validity. However, this is a very time-consuming and tedious task and can only be done for a limited number of traces and gives no guarantee that the considered process models are equal. Also when the process models differ, it might be interesting to work out their common properties and differences and quantify them.

This paper is a continuation of our previous work [9] that determines an upper bound proof for the trace length for comparing two Declare process models for equality based on traces. This approach is mainly simulation-based. It simulates all traces with length lower than or equal to an upper bound and compares them. In this paper, we propose an alternative to this simulation-based approach that completely relies on automata theory. This latter approach has the advantage that the computational effort for simulating traces can be neglected. This is a decisive advantage since this effort might be rather high when complex process models have to be treated. In Sect. 4.5, we recommend how to combine both approaches, the simulation-based and the theory-based, in order to enhance the applicability of our work. We show that both approaches complement each other ideally. Furthermore, in this paper we propose some measures to quantify the differences of non-equal Declare process models.

The remainder of the paper is structured as follows: Section 2 recalls basic terminology, explains the necessary foundations of automata theory and introduces a running example. In Sect. 3, we give an overview of related work and show how our work differs from existing work. In Sect. 4, we recall the simulation-based approach from [9] and propose its advanced version. Additionally we introduce some measures which help to measure up the similarity of Declare models. Section 5 presents the implementation, discusses the asymptotic behavior of the proposed algorithms and presents a practical application of our approach. Finally, Sect. 6 concludes the work and gives an outlook on future work.

2 Basic terminology and running example

In this section, we recall basic terminology and the foundations of automata theory and introduce a running example. Events, traces and event logs are introduced to provide a common basis for the contents of both process models and process traces. Afterward, we give a short introduction of the Declare modeling language, since we focus on this modeling language in the rest of the paper. We also introduce the foundations of automata theory, since our approach is founded on this theory.

2.1 Events, traces and event logs

We briefly recall the standard definitions of events, traces and (process) event logs as defined in [10]. We start with the definition of activities and events:

Definition 1 An **activity** is a well-defined step in a business process. An **event** is the occurrence of an activity in a particular process instance.

This definition enables the definition of a trace, which is a time-ordered sequence of events:

Definition 2 Let $\mathcal{E}$ be the universe of all events, i.e., the set of all possible events. A **trace** is a finite sequence $\sigma = \langle e_1, ..., e_n \rangle$ such that all events belong to the same process instance and are ordered by their execution time, where $n{:=}|\sigma|$ denotes the **trace length** of σ.

We say that a trace is completed if the process instance was successfully closed, i.e., the trace does not violate a constraint of the process model and no additional events related to this process instance will occur in future. Note that in case of declarative process modeling languages like Declare the user must stop working on the process instance in order to close it, whereas in imperative process models this is achieved automatically by reaching an end event [3]. However, a process instance can only be closed if and only if no constraint of the underlying process model is violated [3]. From the definitions above, we can derive the definition of an event log.

Definition 3 An **event log** is a finite set $\{\sigma_1, ..., \sigma_n\}$ of completed traces $\sigma_1, ..., \sigma_n$.

2.2 Declare and Declare constraints

Declare is a single-perspective declarative process modeling language that was introduced in [3]. Instead of modeling all viable paths explicitly, Declare describes a set of constraints applied to activities that must be satisfied throughout the whole process execution. Hereby, the control flow and the ordering of the activities are implicitly specified. Each process execution, which does not violate any of the constraints, is a valid execution. Declare constraints are instances of templates, i.e., patterns that define parameterized classes of properties [4]. Each template corresponds to a graphical representation in order to make the model more understandable to the user. Table 1 summarizes the common Declare

Table 1 Semantics for Declare constraints in LTL$_f$

Template	LTL$_f$ semantics
existence(A, n)	$\mathbf{F}(A \wedge \mathbf{X}(\text{existence}(A, n-1)))$, existence$(A, 1) = \mathbf{F}(A)$
absence(A, n)	$\mathbf{G}(\neg A \vee \mathbf{X}(\text{absence}(A, n-1)))$, absence$(A, 0) = \mathbf{G}(\neg A)$
init(A)	A
last(A)	$\mathbf{G}(\neg A \rightarrow \mathbf{F}(A))$
respondedExistence(A, B)	$\mathbf{F}(A) \rightarrow \mathbf{F}(B)$
response(A, B)	$\mathbf{G}(A \rightarrow \mathbf{F}(B))$
alternateResponse(A, B)	$\mathbf{G}(A \rightarrow \mathbf{X}(\neg A \mathbf{U} B))$
chainResponse(A, B)	$\mathbf{G}(A \rightarrow \mathbf{X}(B)) \wedge$ response(A, B)
precedence(A, B)	$\mathbf{F}(B) \rightarrow ((\neg B)\mathbf{U} A)$
alternatePrecedence(A, B)	precedence$(A, B) \wedge \mathbf{G}(B \rightarrow \mathbf{X}($precedence$(A, B))$
chainPrecedence(A, B)	precedence$(A, B) \wedge \mathbf{G}(\mathbf{X}(B) \rightarrow A)$
succession(A, B)	response$(A, B) \wedge$ precedence(A, B)
chainSuccession(A, B)	$\mathbf{G}(A \leftrightarrow \mathbf{X}(B))$
alternateSuccession(A, B)	alternateResponse$(A, B) \wedge$ alternatePrecedence(A, B)
notRespondedExistence(A, B)	$\mathbf{F}(A) \rightarrow \neg\mathbf{F}(B)$
notResponse(A, B)	$\mathbf{G}(A \rightarrow \neg\mathbf{F}(B))$
notPrecedence(A, B)	$\mathbf{G}(F(B) \rightarrow \neg A)$
notChainResponse(A, B)	$\mathbf{G}(A \rightarrow \neg\mathbf{X}(B))$
coExistence(A, B)	$\mathbf{F}(A) \leftrightarrow \mathbf{F}(B)$
notCoExistence(A, B)	$\neg(\mathbf{F}(A) \wedge \mathbf{F}(B))$
choice(A, B)	$\mathbf{F}(A) \vee \mathbf{F}(B)$
exclusiveChoice(A, B)	$(\mathbf{F}(A) \vee \mathbf{F}(B)) \wedge \neg(\mathbf{F}(A) \wedge \mathbf{F}(B))$

templates. Although Declare provides a broad repertoire of different templates, which covers the most necessary scenarios, this set is non-exhaustive and can be arbitrarily extended by the modeler defining new templates. Hence, the user is not aware of the underlying logic-based formalization that defines the semantic of the templates (respectively constraints). Declare relies on the linear temporal logic (LTL) over finite traces (LTL$_f$) [3]. Hence, we can define a Declare process model formally as follows:

Definition 4 A **Declare process model** is a pair $(A, \mathcal{T})$ where A is a finite set of activities and $\mathcal{T}$ is a finite set of LTL constraints over A (i.e., instances of the predefined templates or LTL formulas).

LTL makes it possible to define conditions or rules about the future of a system. In addition to the common logical connectors ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) and atomic propositions, LTL provides a set of temporal (future) operators. Let ϕ_1 and ϕ_2 be LTL formulas. The future operators $\mathbf{F}$, $\mathbf{X}$, $\mathbf{G}$, $\mathbf{U}$ and $\mathbf{W}$ have the following meaning: formula $\mathbf{F}\phi_1$ means that ϕ_1 sometimes holds in the future, $\mathbf{X}\phi_1$ means that ϕ_1 holds in the next position, $\mathbf{G}\phi_1$ means that ϕ_1 holds forever in the future and $\phi_1\mathbf{U}\phi_2$ means that sometimes in the future ϕ_2 will hold and until that moment ϕ_1 holds. The weaker form of the until operator ($\mathbf{U}$), the so-called weak until $\phi_1\mathbf{W}\phi_2$ has the same meaning as the until operator, whereby ϕ_2 is not required to hold. In this case, ϕ_1 must hold forever.

For a more convenient specification, LTL is often extended to past linear temporal logic (PLTL) [11] by introducing so-called past operators, which make it possible to define conditions or rules about the past but do not increase the expressiveness of the formalism [12]. The past operators $\mathbf{O}$, $\mathbf{Y}$ and $\mathbf{S}$ have the following meaning: $\mathbf{O}\phi_1$ means that ϕ_1 sometimes holds in the past, $\mathbf{Y}\phi_1$ means that ϕ_1 holds in the previous position and $\phi_1\mathbf{S}\phi_2$ means that ϕ_1 has held sometimes in the past and since that moment ϕ_2 holds.

For a better understanding, we exemplarily consider the response constraint $\mathbf{G}(A \rightarrow \mathbf{F}B)$. This constraint means that if A occurs, B must eventually follow sometimes in the future. We consider, for example, the following traces: $T_1 = \langle A, A, B, C\rangle$, $T_2 = \langle B, B, C, D\rangle$, $T_3 = \langle A, B, C, B\rangle$ and $T_4 = \langle A, B, A, C\rangle$. The traces T_1, T_2 and T_3 satisfy the response constraint as each occurrence of activity A is followed by an occurrence of activity B. Note that T_2 fulfills this constraint trivially because activity A does not occur at all (so-called *vacuously satisfied*). However, T_4 violates the constraint, because after the second occurrence of A no execution of B follows.

We say that an event activates a constraint in a trace if its occurrence imposes some obligations on other events in the same trace. Such an activation leads either to a fulfillment or to a violation of a constraint. Consider, for example, the response constraint response(A, B). This constraint is activated by the execution of activity A. In T_4, for instance, the response constraint is activated twice. In case of the first activation, this leads to a fulfillment because B occurs. However, the second activation leads to a violation because B does not occur subsequently.

In our research, we use Declare as a representative for declarative process modeling languages. Declare is rather prominent in the process modeling community and is investigated thoroughly what is supporting our decision. In principle, our approach would allow to exchange the declarative process modeling language. In order to do so, the language constructs of that language would have to be transformed to finite state automatons as will be shown in Sect. 4.1. Having settled this transformation, our methods can further be applied as shown in this paper.

2.3 Automata theory

Our approach is mainly based on deterministic finite state automatons (FSA). We aim to map the underlying Declare process models to finite state automatons in order to extract information, which can be used to make statements about the process models. Therefore, we briefly introduce the basic concepts and algorithms of automata theory. For further details cf. [13]. We start with the formal definition of a deterministic finite state automaton:

Definition 5 A **deterministic finite-state automaton (FSA)** is a quintuple $M = (\Sigma, S, s_0, \delta, F)$ where Σ is a finite (non-empty) set of symbols, S is a finite (non-empty) set of states, $s_0 \in S$ is an initial state, $\delta : S \times \Sigma \to S$ is the state-transition function and $F \subseteq S$ is the set of final states.

As we want to deal with words and not only single symbols, we have to expand the definition:

Definition 6 Let Σ be a finite (non-empty) set of symbols. Then, $\Sigma^*{:=}\{a_1 a_2 \dots a_n \mid n \in \mathbb{N}_0, a_i \in \Sigma\}$ is the **set of all words** over symbols in Σ. For each word $\omega \in \Sigma^*$, we define the **length of** ω as

$$|\omega|{:=}\begin{cases} 0 & \omega = \varepsilon \ (\varepsilon \text{ denotes the empty string}),\\ 1 & \omega \in \Sigma,\\ |a| + |b| & \omega = ab \text{ with } a \in \Sigma \text{ and } b \in \Sigma^*. \end{cases}$$

Definition 7 For a FSA $M = (\Sigma, S, s_0, \delta, F)$, we define the **extended state-transition function** $\hat{\delta} : S \times \Sigma^* \to S$,

$$(s, \omega) \mapsto \begin{cases} s & \omega = \varepsilon,\\ \delta(s, \omega) & \omega \in \Sigma,\\ \delta(\hat{\delta}(s, a), b) & \omega = ab \text{ with } a \in \Sigma \text{ and } b \in \Sigma^*. \end{cases}$$

In the following, for the sake of simplicity, δ always denotes the extended state-transition function $\hat{\delta}$ for words $\omega \in \Sigma^*$. The set of words that are accepted by a FSA M is called the language of M:

Definition 8 Let $M = (\Sigma, S, s_0, \delta, F)$ be a FSA. Then, $\mathcal{L}(M){:=}\{\omega \in \Sigma^* \mid \delta(s_0, \omega) \in F\} \subseteq \Sigma^*$ is called the **language of** M.

Words that can be constructed from the same alphabet, but are not accepted by the FSA, form the complement of the language:

Definition 9 Let $\mathcal{L}(M)$ be the language of a FSA $M = (\Sigma, S, s_0, \delta, F)$. Then, $\mathcal{L}(M)^C{:=}\Sigma^* \backslash \mathcal{L}(M)$ is called the **complement of** $\mathcal{L}(M)$.

Remark 1 For a finite state automaton $M = (\Sigma, S, s_0, \delta, F)$, an automaton that accepts exactly the complement of $\mathcal{L}(M)$ can be obtained by swapping its final states with its non-final states and vice versa [13]. This automaton M^C is called the *complementary automaton of* M.

Example 1 Consider $\Sigma = \{A, B\}$. Then,

$$\Sigma^* = \{\epsilon, A, B, AA, AB, BA, BB, ABB, \dots\}$$

consists of all words including any number of As and Bs. The set $L{:=}\{A\omega \mid \omega \in \Sigma^*\} = \{A, AA, AB, AAA, \dots\}$ is the language of all words with A at the beginning. The corresponding FSA is depicted on the left side of Fig. 1. The complement L^C of L consists of all words of Σ^* which do not start with A: $L^C = \{\omega_1\omega_2 \mid \omega_1 \in \Sigma\backslash\{A\}, \omega_2 \in \Sigma^*\}$. The corresponding automaton is illustrated on the right side of Fig. 1.

As we have to handle with automatons that consist of a big number of states, it is desirable to decrease the number of states in order to improve the performance. In general, there exists a minimal automaton which accepts the same language:

Theorem 1 *Let $M = (\Sigma, S, s_0, \delta, F)$ a finite state automaton. Then there exists a minimal (based on the number of states) finite state automaton $M_{\min} = (\Sigma, S_{\min}, s_0, \delta_{\min}, F_{\min})$ with $\mathcal{L}(M_{\min}) = \mathcal{L}(M)$.*

Proof cf. [13]. □

Remark 2 This theorem is trivially fulfilled if M is already minimal. If M is not minimal, we use the Hopcroft algorithm [14] to construct an equivalent minimal finite state automaton.[1]

[1] We decided for this algorithm, because it is the fastest algorithm (time complexity $\mathcal{O}(n \log \log n)$ where n denotes the number of states of the FSA) for this task.

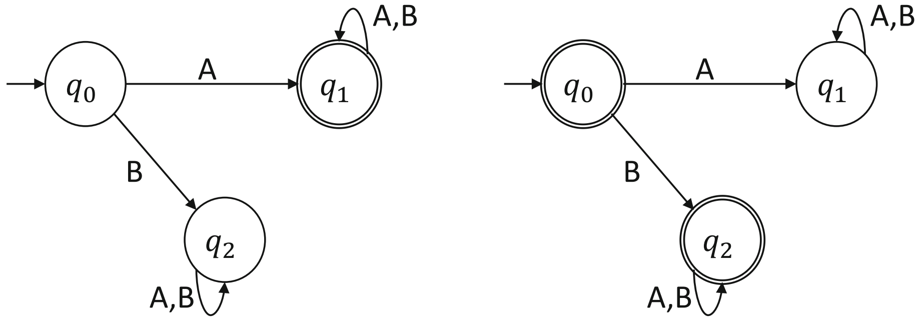


Fig. 1 Finite state automaton M with $\mathcal{L}(M) = \{A\omega \mid \omega \in \{A, B\}^*\}$ and complementary finite state automaton M^C with $\mathcal{L}(M)^C = \{\omega_1\omega_2 \mid \omega_1 \in \Sigma\backslash\{A\}, \omega_2 \in \Sigma^*\}$

Given two FSAs, we are interested in the intersection of their corresponding languages, i.e., the set of all words that are accepted by both. Therefore, we can use the construct of the product automaton:

Definition 10 Let $M_1 = (\Sigma, S_1, s_{0_1}, \delta_1, F_1)$ and $M_2 = (\Sigma, S_2, s_{0_2}, \delta_2, F_2)$ two deterministic finite-state automatons over the same set of symbols Σ. The product automaton $M = M_1 \times M_2$ is defined as the quintuple $M = (\Sigma, S_M, s_{0_M}, \delta_M, F_M)$ where $S_M = S_1 \times S_2$, $s_{0_M} = (s_{0_1}, s_{0_2})$, $\delta_M : S \times \Sigma \to S, ((s_1, s_2), a) \mapsto (\delta_1(s_1, a), \delta_2(s_2, a))$ and $F_M = F_1 \times F_2$.

From the definition of the product automaton $M = M_1 \times M_2$ of two deterministic finite-state automatons M_1 and M_2 follows that M accepts exactly the intersection of $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ [13]: $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.

Furthermore, an automaton for the symmetric product of two automatons can be calculated: given two finite state automatons M_1 and M_2, an automaton that accepts exactly the words of M_1 (and not by M_2) can be constructed by calculating the automaton $M_1 \times M_2^C$, i.e., $\mathcal{L}(M_1 \times M_2^C) = \mathcal{L}(M_1)\backslash\mathcal{L}(M_2)$. We will use this construction later in our approach.

2.4 Running example

In the following, we will refer extensively to the following two examples, which reflect the different application scenarios of our approach.

Example 1 The first sample process P consists of a set $\mathcal{A}$ of three activities A, B and C with the following control flow: either the three activities are executed in sequence (i.e., ABC) or alternatively C is executed arbitrarily often but at least once. After each execution of the sequence ABC also the sequence BC can be executed arbitrarily often. The Declare language offers manifold ways for modeling this process. For example, we can describe this process by the following two process models:

- $P_1 = (\mathcal{A}, \mathcal{T}_1)$, with $\mathcal{T}_1 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ containing the following constraints: t_1: response(A, B), t_2: precedence(A, B), t_3: respondedExistence(A, B), t_4: response(A, C), t_5: notChainResponse(B, B), t_6: $\mathbf{G}\,(A \to \mathbf{X}(B) \wedge \mathbf{X}(\mathbf{X}(C)))$ and t_7: $\mathbf{G}(B \to \mathbf{X}(\neg A))$.
- $P_2 = (\mathcal{A}, \mathcal{T}_2)$, with $\mathcal{T}_2 = \{t_1^{'}, t_2^{'}, t_3^{'}, t_4^{'}\}$ containing the following constraints: $t_1^{'}$: succession(A, B), $t_2^{'}$: chainResponse(A, B), $t_3^{'}$: respondedExistence(A, B) and $t_4^{'}$: chainResponse(B, C).

For a better illustration, the process models are depicted in graphical Declare notation in Fig. 2a, b. Apart from the respondedExistence template that occurs in both process models, P_1 and P_2 seem to be completely different. Hence, it is difficult to assess, whether the two process models really describe the same process. We will show throughout the paper, how our approach can be used to validate this claim.

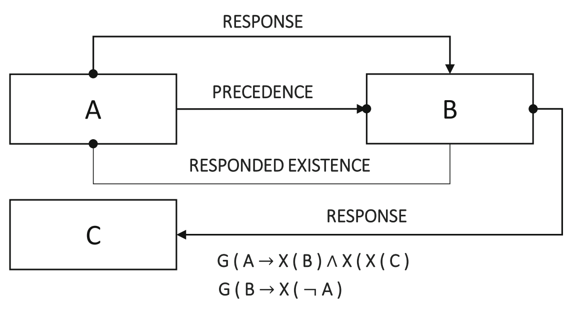
Example 2 In the second scenario, we consider two process models $Q_1 = (\{A, B, C\}, \mathcal{S}_1)$ and $Q_2 = (\{A, B, C\}, \mathcal{S}_2)$ where

$$\mathcal{S}_1 = \{\text{existence(A,1), exclusiveChoice(A,B)}\} \text{ and } \mathcal{S}_2 = \{\text{existence(B, 2)}\}.$$

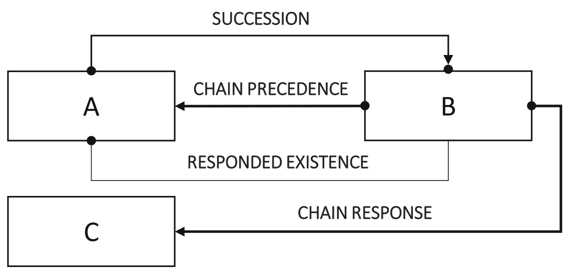
Obviously, these process models describe different processes, since activity A has to be executed at least once in Q_1, whereas in Q_2 it does not have to be executed. Hence, Q_2 accepts the trace $\langle BB \rangle$ (as the only constraint of Q_2 demands for a double execution of activity B) and Q_1 does not because an execution of activity A is missing. We will use this example in order to demonstrate how our approach can be used for analyzing differences of Declare process models. Both process models are depicted in graphical Declare notation in Fig. 3a, b.

3 Related work on process model similarity

Determining similarity and common properties of process models is a very important issue in industry and research [15,16]. It is on the one hand necessary to identify duplicate models [17] and different model variants [18], which might be produced when process models are changed or

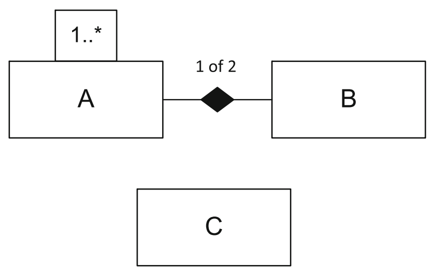


(a) Process Model P_1

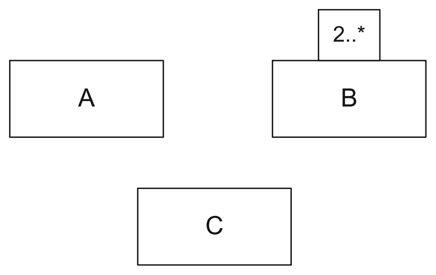


(b) Process Model P_2

Fig. 2 Graphical ConDec representation of Declare models P_1 and P_2



(a) Process Model Q_1



(b) Process Model Q_2

Fig. 3 Graphical ConDec representation of Declare models Q_1 and Q_2

emerged. This work relates to the stream of research on modeling and checking declarative process models. Difficulties in understanding and modeling declarative processes are a well-known problem in the current research. Nevertheless, there are only a handful of experimental studies that deal with the understandability of declarative process models. In [19], a study reveals that single constraints can be handled well by most individuals, whereas sets of constraints establish a serious challenge, i.e., Declare models consisting of more than a handful constraints might get very hard or even not at all understandable for humans. Furthermore, it has been stated that individuals use the composition of the notation elements for interpreting Declare models. Similar studies [8,20] investigated the understandability of hybrid process representations which consist of graphical- and text-based specifications.

For different model checking tasks of both multi-perspective and single-perspective declarative process models, there are different approaches. In [21], an automaton-based approach is presented for the detection of redundant constraints and contradictions between the constraints, which does not fill the gap of different process models on equality or differences. In [22,23], the problem of the detection of hidden dependencies is addressed. Hidden dependencies are dependencies between activities which are not modeled explicitly but result of the combination of certain different constraints. In [22], the extracted hidden dependencies are added to the Declare models through visual and textual annotations to improve the understandability of the models. In [24], the authors transform the common Declare templates in a standardized form called positive normal-form, with the aim of simplifying model comparisons. But also this approach reaches its limits because the positive normal-form is not unique, and hence, different positive normal-forms can describe the same model. The authors in [25] investigate the single elements of process models in order to detect corresponding or equivalent elements in different process models. Hence, equivalent elements, e.g., activities or actors, can be identified but there is still the need of combining all these elements as they represent a whole process model.

There is also some effort in transforming Declare process models into different representations for deeper analysis. In [26], formulas of linear temporal logic over finite traces are translated to both nondeterministic and deterministic finite automatons, which were not investigated yet in order to compare the underlying process models. In [27] Büchi automatons are generated from LTL formulas. In [28], Declare templates are translated into deterministic finite automatons, which are used for implementing a declarative discovery algorithm for the Declare language. Also these efforts do not deliver a possibility to compare the process models.

The standard procedure for comparing the desired behavior with the expected behavior provided in a process model includes the generation of exemplary process executions [29], which are afterward analyzed in detail with regard to undesired behavior such as contradictions, deadlocks or deviations from the behavior in reality. Therefore, process execution traces up to a certain length are simulated and investigated. This procedure has the weakness that the calculation has to be stopped at some trace length due to computing power and storage requirements. Hence, a 100% statement about equality or inequality cannot be manifested as there might be undetected inconsistences in traces of larger lengths. In [9], we handled this issue by computing a theoretical upper bound for the trace length in order to make it possible to decide about equality with certainty. The underlying paper extends this method by presenting an alternative approach for the automaton comparison and giving measures, which help to make statements about differences of non-equal process models.

In [30], the authors define the equality between two process models (regardless of whether they are imperative or declarative) on the base of all viable process execution paths. Often, for a better understanding of a model, also counterexamples are explicitly constructed to verify whether a model prevents a particular behavior [31]. For generating exemplary process executions, it is necessary to execute declarative process models. In [32], both MP-Declare templates and Declare templates are translated into the logic language Alloy[2] and the corresponding Alloy framework is used for the execution. For generating traces directly from a declarative process model (i.e. MP-Declare as well as Declare) the authors in [33].

In [31], based on a given process execution trace (that can also be empty), possible continuations of the process execution are simulated up to an a-priori defined length. The authors emphasize the usefulness of model checking of (multi-perspective) declarative processes by simulating different behavior. However, the length of the look-ahead is chosen arbitrarily and, hence, can only guarantee the correctness of a model up to a certain trace length. In summary, the need for a generally applicable algorithm to determine the minimum trace length required to find out whether process models are equivalent is still there, and this issue has not been solved so far.

4 Comparing Declare process models

In this introductory part, we want to give a brief overview on our approach. Details about the single steps for comparing Declare process models will be explained in the corresponding subsections. The overall concept is illustrated in Fig. 4.

[2] https://alloytools.org/.

The input of our approach are two Declare models $P_1 = (\mathcal{A}_1, \mathcal{T}_1)$ and $P_2 = (\mathcal{A}_2, \mathcal{T}_2)$. In a preparation phase (cf. Sect. 4.1), we first transform each template of the Declare models into deterministic finite state automatons (step 1 in Fig. 4). Afterward, we construct (minimal) FSAs D_1 and D_2 for each process model by intersecting the automatons of the corresponding templates (step 2 in Fig. 4, cf. Sect. 4.2).

After calculating the two product automatons D_1 and D_2, we can apply our comparison algorithms (step 3 in Fig. 4), i.e., we compare the two automatons with respect to equality. Firstly, this is done by comparing all words of an automaton until a particular length (so-called simulation-based approach) that guarantees whether D_1 and D_2 are equal (cf. Sect. 4.3.1). Secondly, as an alternative approach (so-called theory-based approach[3]) the comparison takes exclusively place by directly investigating the automatons themselves (cf. Sect. 4.3.2). The simulation-based approach and the theory-based approach complement each other. Thus, we provide a short recommendation when to apply what algorithm in Sect. 4.5.

If the process models are equivalent there is no further work to do, otherwise we analyze their differences in detail (cf. Sect. 4.4). This encompasses checking the models for mutual containment (step 4 in Fig. 4) and calculating the intersection and differences of the process models (step 5 in Fig. 4).

The resulting automatons of intersection and differences are often difficult to interpret and compare. So it is a common approach to generate traces of certain lengths that are accepted by the automatons and analyze and compare those sets. Therefore, we propose and apply some measures to quantify the differences of the process models (cf. Sect. 4.6).

4.1 Transformation of Declare templates to finite state automatons

The first step of our approach is to transform Declare templates into deterministic finite state automatons (step 1 Fig. 4). For the most common Declare templates, this transformation was already done in [28]. However, the Declare templates notRespondedExistence and notResponse are not dealt with in that paper; their representations as FSAs are shown in Figs. 5 and 6. Traces fulfilled by a Declare template are exactly the elements of the accepted language of the corresponding FSA. For example, trace $\sigma_1 = \langle A, A \rangle$ fulfills the notResponse template, whereas trace $\sigma_2 = \langle A, A, B \rangle$ does not. The same thing holds for the automaton, too: σ_1 is accepted and σ_2 is not accepted by the corresponding automaton (cf. Fig. 6).

[3] We call this approach ‘theory-based’ since it is solely grounded in automata theory.

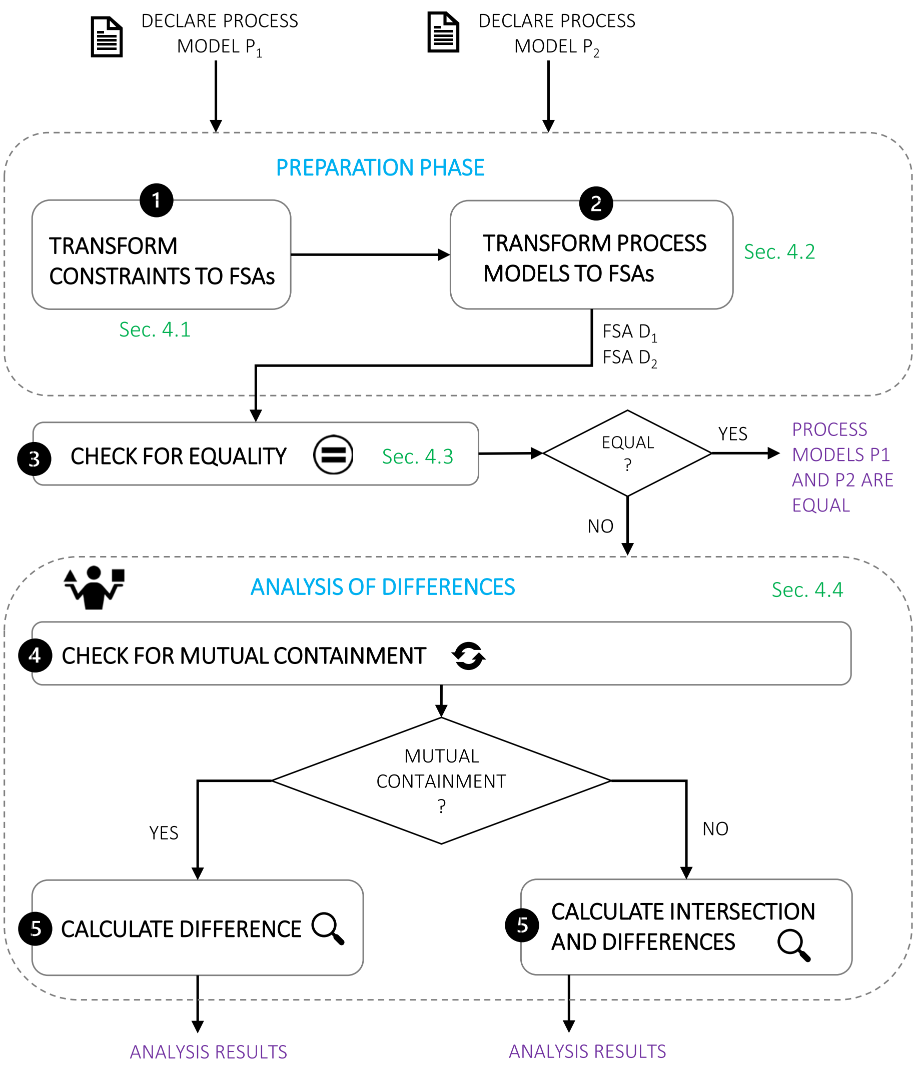

DECLARE PROCESS MODEL P$_1$
DECLARE PROCESS MODEL P$_2$
PREPARATION PHASE
1 TRANSFORM CONSTRAINTS TO FSAs
Sec. 4.1
2 TRANSFORM PROCESS MODELS TO FSAs
Sec. 4.2
FSA D$_1$
FSA D$_2$
3 CHECK FOR EQUALITY
Sec. 4.3
EQUAL ?
YES
PROCESS MODELS P1 AND P2 ARE EQUAL
NO
ANALYSIS OF DIFFERENCES
Sec. 4.4
4 CHECK FOR MUTUAL CONTAINMENT
MUTUAL CONTAINMENT ?
YES
NO
5 CALCULATE DIFFERENCE
5 CALCULATE INTERSECTION AND DIFFERENCES
ANALYSIS RESULTS
ANALYSIS RESULTS

Fig. 4 Procedure for Declare model comparison

In a Declare model, multiple activities are involved within multiple templates. One concrete template normally comprises one or two activities. From the viewpoint of such a template, we have to consider those activities that are associated with other activities as well. Since their executions do not have an impact on the execution of the template under considerations, we add transitions of type *:otherwise* to the corresponding automaton. These transitions represent all activity executions of activities that do not occur in the respective template. For example, in Fig. 6 the Declare template defines a dependency between activities A and B. When A and B are occurring, respective, state transitions are initiated. Nevertheless, this template might be part of a comprehensive process model that also contains the activities C, D, and E. Referring to the template from Fig. 6, whenever these three activities are occurring they are “swallowed” by the transitions: *otherwise*, i.e., they do not change the state of the FSA.

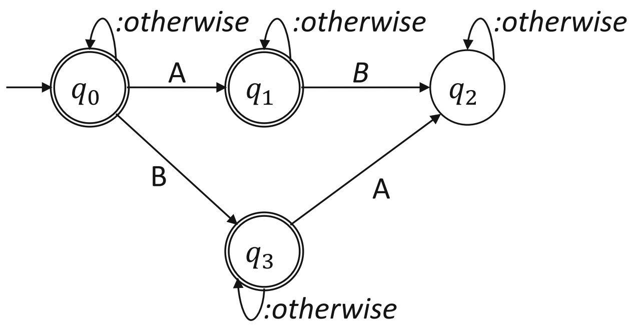


Fig. 5 notRespondedExistence(A, B)

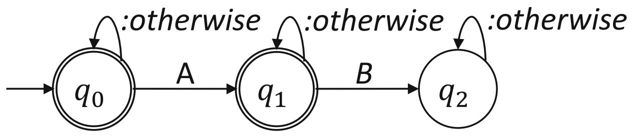


Fig. 6 notResponse(A, B)

4.2 Transformation of Declare models to finite state automatons

Next, Declare process models have to be transformed into finite state automatons (step 2, Fig. 4). This procedure is described in Algorithm 2. Hence, the process model is represented by a deterministic finite state automaton and the words of the automatons correspond to the valid traces of the process model (i.e., the language of the automaton is the set of all valid traces).

As a Declare process model, P consists of a set of different Declare templates $\mathcal{T} = \{t_1, \dots, t_n\}$, a trace σ that satisfies P is a trace, that satisfies all templates:

$$\sigma \text{ satisfies } P \iff \sigma \text{ satisfies } t_1 \wedge \dots \wedge t_n \tag{1}$$

By using the concept of the product automaton (cf. Sect. 2.3), the following conclusion can be derived:
A trace σ satisfies a Declare model $P = (\mathcal{A}, \mathcal{T})$ if and only if $\sigma \in \mathcal{L}(M_1) \cap \dots \cap \mathcal{L}(D_n) = \mathcal{L}(D_1 \times \dots \times D_n)$ where D_i is the corresponding FSA of t_i.

Remark 3 The resulting product automaton $D_1 \times \dots \times D_n$ consists of $|S_1| \cdot \dots \cdot |S_n|$ states. In order to potentially decrease the number of states, we can use the Hopcroft minimalization algorithm [14] after each intersection of two automatons D_i and D_j. This means that the minimization algorithm will be called $n - 1$ times during the calculation.

Note that the minimization algorithm does not change the effectiveness of our approach. Minimization just helps to decrease the number of states in order to reduce storage space needed for our computations and to speed up the algorithm.

4.3 Checking Declare models for equality

Based on the previous results, it is now possible to construct algorithms for checking equality of two Declare process models $P_1 = (\mathcal{A}_1, \mathcal{T}_1)$ and $P_2 = (\mathcal{A}_2, \mathcal{T}_2)$ (step 3 in Fig. 4). Therefore, we check the corresponding finite state automatons for equality, i.e., we check whether they accept the same language. This can either be achieved by considering the words until a particular length accepted by the automatons which guarantees to decide the equality of the automatons (simulation-based approach). This approach was already proposed in our previous work [9]. Alternatively we can check the equality by directly investigating the automatons themselves (theory-based approach), which is one of the new contribution of this article. In the following, we describe the two approaches. Note that both approaches are also applicable for checking more than two models for equality: respectively two models can be compared in pairs in order to get information about more models.

4.3.1 Simulation-based approach

This approach constructs traces of a particular length and compares them. The essential part of the simulation-based approach is to determine an upper bound, i.e., a maximal trace length until which the traces must be simulated in order to decide with certainty whether two Declare models are equal. Therefore, we formulate and prove a theorem that determines this upper bound.

Theorem 2 *Let D_1 and D_2 be two FSAs with m states and n states. Then, $\mathcal{L}(D_1) = \mathcal{L}(D_2)$ if and only if $\{\omega \in \mathcal{L}(D_1) \mid |\omega| < mn\} = \{\omega \in \mathcal{L}(D_2) \mid |\omega| < mn\}$*

Proof We prove the two directions of the implication. As $\mathcal{L}(D_1) = \mathcal{L}(D_2)$, the equality holds for all subsets. That implies especially that $\{\omega \in \mathcal{L}(D_1) \mid |\omega| < mn\} = \{\omega \in \mathcal{L}(D_2) \mid |\omega| < mn\}$.

We prove the opposite direction by contrapositive. So suppose $\mathcal{L}(D_1) \neq \mathcal{L}(D_2)$ and let a be a word of minimal length with $a \notin \mathcal{L}(D_1) \cap \mathcal{L}(D_2) = \mathcal{L}(D_1 \times D_2)$. We further define $D{:=}D_1 \times D_2$ as the product automaton of D_1 and D_2.

We assume by contradiction that $|a| \geq mn$. We define $X{:=}\{\delta(q_0, b) \mid b \text{ prefix of } a\}$. Since $|X| \geq mn + 1$ and $|S_D| = mn$, there exist two prefixes u and u' of a with $\delta_D(q_0, u) = \delta_D(q_0, u')$. We assume without any loss of generality that u is a prefix of u'. So there are two words v and z so that $uv = u'$ and $u'z = a$. It follows that $uvz = a$.

As $u \neq u'$, v is not empty. The equation $\delta_D(\delta_D(q_0, u), v) = \delta_D(q_0, u)$ says that v leads D through a loop from state $\delta_D(q_0, u)$ into itself. So we have found a word uz with $\delta_D(q_0, uz) = \delta_M(q_0, a)$ with $|uz| < |a|$. This is a contradiction to the minimality of a. □

□

The interpretation of this theorem for our purposes is the following: in order to check the underlying Declare models for equality we have to calculate the upper bound $b{:=}|D_1| \cdot |D_2|$ where $|D_i|$ denotes the number of states of automaton D_i. Afterward, all words up to length b have to be simulated and checked whether D_1 and/or D_2 accepts them (this can be done via a trace generator for Declare process models [32,33] or by deriving all words directly from the automaton). If they both accept the same set of words, i.e., $\{\omega \mid |\omega| \leq b \text{ and } D_1 \text{ accepts } \omega\} = \{\omega \mid |\omega| \leq b \text{ and } D_2 \text{ accepts } \omega\}$, the automatons are equal. Otherwise, they are not equal.

4.3.2 Theory-based approach

Instead of simulating traces, the theory-based approach deals exclusively with the automatons themselves and does not require the simulation of traces. This procedure is illustrated in Algorithm 1. The algorithm returns *true* if two FSAs are equivalent and *false* otherwise. At first, the two automatons are minimized (if not already done), afterward we calculate the symmetric product D of the two automatons D_1 and D_2 (cf. Alg. 1, line 3) as described in Sect. 2.3. If the resulting automaton has no accepting states (cf. Alg. 1, line 4–5), the FSAs are equivalent.

Algorithm 1: *areEqual*

```
Input: Finite state automatons $D_1 = (\Sigma, S_1, s_{0_1}, \delta_1, F_1)$ and
       $D_2 = (\Sigma, S_2, s_{0_2}, \delta_2, F_2)$
Output: True if the automatons are equivalent, otherwise False
1 $D_1 \leftarrow$ minimize($D_1$)
2 $D_2 \leftarrow$ minimize($D_2$)
3 $D \leftarrow$ symmetricDifferenceProduct($D_1$, $D_2$)
4 if $F_D = \emptyset$ then
5     return True
6 else
7     return False
8 end
```

Algorithm 2: *calculateProductAutomaton*

```
Input: Declare Process Model $P = (\mathcal{A}, \mathcal{T})$
Output: Representing FSA
1 $U \leftarrow \emptyset, i \leftarrow 1$
2 for $t \in \mathcal{T}$ do
3     $(A, S_t, s_{0_t}, \delta_t, F_t) \leftarrow$ transform $t$ to minimal FSA
4     $t_i \leftarrow (A, S_t, s_{0_t}, \delta_t, F_t)$
5     $U \leftarrow U$.add($(A, S_t, s_{0_t}, \delta_t, F_t)$)
6     $i \leftarrow i + 1$
      /* Calculating set $U$ of automatons for $\mathcal{T}$
         */
7 end
8 for $j = 1, \ldots, i - 1$ do
9     $t_{j+1} \leftarrow$ minimization(product($t_j$, $t_{j+1}$))
      /* Calculating minimal product automaton
         of $P$                                  */
10 end
11 $D \leftarrow t_{j+1}$
12 return $D$
```

4.3.3 Example

We now apply the two approaches for checking equality to the process models P_1 and P_2 of the running example. First, we must transform each constraint of the process models into a finite state automaton. Afterward, the automatons of the constraints of each process model are intersected and minimized, to represent each process model as a single automaton. Calculating the automatons D_1 and D_2 leads to the same automaton, which is depicted in Fig. 7. Note that the minimization step has an impressive effect: Without minimization, the product automatons have $|S_{t_1}| \cdot |S_{t_2}| \cdot |S_{t_3}| \cdot |S_{t_4}| \cdot |S_{t_5}| \cdot |S_{t_6}| \cdot |S_{t_7}| = 1296$ states (process model

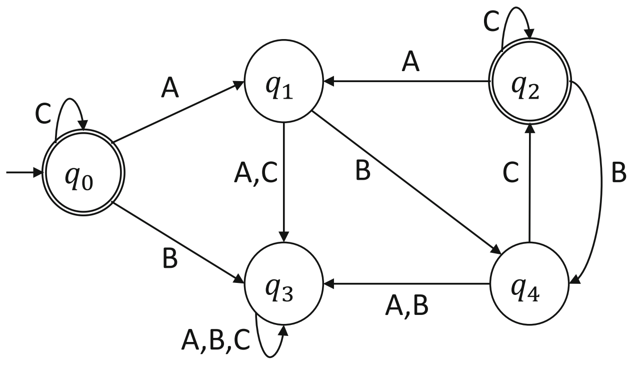

$$\begin{tikzcd}
 & q_1 \arrow[rd, "B"] \arrow[d, "{A,C}"] & q_2 \arrow[l, "A"'] \arrow[loop above, "C"] \arrow[d, bend left, "B"] \\
\arrow[r] q_0 \arrow[loop above, "C"] \arrow[ru, "A"] \arrow[r, "B"'] & q_3 \arrow[loop below, "{A,B,C}"] & q_4 \arrow[l, "{A,B}"] \arrow[u, "C"]
\end{tikzcd}$$

Fig. 7 Minimized automaton for P_1 and P_2

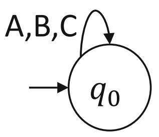

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "{A,B,C}"]
\end{tikzcd}$$

Fig. 8 Symmetric difference of the automatons of P_1 and P_2

P_1) and $|S_{t'_1}| \cdot |S_{t'_2}| \cdot |S_{t'_3}| \cdot |S_{t'_4}| = 108$ states (process model P_2), respectively, since the number of states of the product automatons is the product of the numbers of states of the single template automatons. The minimized automatons only contain 5 states (cf. Fig. 7). In case of the simulation-based approach, Theorem 2 tells us that we must simulate all traces up to length $|S_1| \cdot |S_2| = 5 \cdot 5 = 25$ (without minimization it would be necessary to consider all traces up to a length of $|S_1| \cdot |S_2| = 1296 \cdot 108 = 139.968$). Generating all traces for both process models until the determined upper bound and comparing them reveals that the two process models are indeed equal. In the theory-based approach it is not necessary to determine an upper bound and simulate traces. It is sufficient to create the (minimal) symmetric difference product of both automatons. The results of this construction are depicted in Fig. 8, which shows that this automaton does not contain an accepting state. Hence, the automatons and thus the process models are identical.

4.4 Analyzing mutual containment and differences between Declare models

In case that two process models are not equal, it is interesting to identify and also to interpret their differences. The following two questions arise:

Q1 Is one model contained within the other one, i.e., $\{\sigma \mid \sigma \text{ satisfies } P_1\} \subset \{\sigma \mid \sigma \text{ satisfies } P_2\}$ (i.e., all traces accepted by one model are accepted by the other one) or vice versa (*Mutual Containment*)?

Q2 What are the common properties of the models, i.e., which traces are accepted by both models and where are the differences between the models, i.e., which traces are accepted by P_1 but not by P_2 and vice versa?

Answer to question 1

Let D_1 be the corresponding finite state automaton of P_1 and D_2 be the corresponding finite state automaton of P_2. For checking mutual containment (step 4 in Fig. 4), we first check whether $\{\omega \mid D_1 \text{ accepts } \omega\} \subset \{\omega \mid D_2 \text{ accepts } \omega\}$, i.e., whether the first Declare model is completely contained in the second one. If the result is *true*, we get the information that $\mathcal{L}(D_1)$ is contained in $\mathcal{L}(D_2)$ and thus P_1 is contained in P_2. Otherwise we check the opposite containment relation, i.e., $\{\omega \mid D_2 \text{ accepts } \omega\} \subset \{\omega \mid D_1 \text{ accepts } \omega\}$. If none of the models is contained in the other one, they describe quite different applications.

For checking the containment of $\mathcal{L}(D_i)$ in $\mathcal{L}(D_j)$, we calculate the product automaton P of D_i and D_j. This automaton accepts exactly the intersection between $\mathcal{L}(D_i)$ and $\mathcal{L}(D_j)$. We check whether the intersection P is equal to D_i. If they are equal, $\mathcal{L}(D_i)$ is a subset of $\mathcal{L}(D_j)$.

Answer to question 2

For answering the second question, we construct automatons describing the intersection $\mathcal{L}(D_2) \cap \mathcal{L}(D_1)$ and the differences $\mathcal{L}(D_1) \backslash \mathcal{L}(D_2)$ and $\mathcal{L}(D_2) \backslash \mathcal{L}(D_1)$ (step 5 in Fig. 4). We calculate the product automaton of D_1 and D_2 in order to get an automaton for the intersection. For calculating the difference $\mathcal{L}(D_i) \backslash \mathcal{L}(D_j)$, we first calculate the product automaton of D_i and the complement of D_j (cf. Sect. 2.3). The resulting automaton accepts the set

$$\{\sigma \mid \sigma \text{ satisfies } P_1 \text{ and } \sigma \text{ does not satisfy } P_2\}.$$

In case the results of the above case analysis—which are automatons that reflect common or different parts of languages—are not illustrative enough, we provide a practical approach to illustrate these partial languages. We simulate traces up to different lengths. These traces present either common or varying parts of the two Declare process models. By producing traces of different lengths, the domain experts get an impression of the common parts and the differences of the Declare process models to be compared. Besides, the generated traces can be analyzed afterward by applying various measurements (cf. Sect. 4.6).

4.5 Simulation-based versus theory-based approach

The simulation-based approach and the theory-based approach lead to equal results from a qualitative perspective: they decide whether two process models are equal or not. Nevertheless, the calculation of the results is quite different and the intermediate results can be used for quite different considerations of the process models. Also the effort to reach results is totally different for both approaches. The simulation-based approach is pretty time- and cost-consuming. However, its results are very illustrative since it delivers concrete process traces that are produced by one or by two process models to be compared. In contrast, comparing two process models, i.e., comparing their corresponding automatons, is quite economical with the theory-based approach. Without huge calculations, similarities and dissimilarities of process models to be compared can be calculated. Albeit, results produced by this approach are kind of abstract since only automatons are produced reflecting the similarities and dissimilarities.

Based on the observations from above, we recommend the following processing. First, we would apply the theory-based approach in order to receive a general overview on the equality of two process models. The main advantage of this proceeding is the low effort this approach is requesting and the clear results concerning similarities and dissimilarities. Depending on further users' interest, the simulation-based approach can be applied afterward. This will add concrete results, i.e., process traces, to the formerly performed theory-based approach and so will illustrate the abstract results of the first approach. Nevertheless, this processing is just a recommendation. Finally, users of our algorithm have to find out their preferred usage that heavily depends on whether they need more or less illustrative feedback and whether they can spend more or less computing time. Although we do not give more than a recommendation how to apply the simulation-based and the theory-based approach, our experience reveals that both approaches complement each other and together provide promising insights into the similarity issue of Declare process models. As stated in the related work, there are no alternative approaches in literature that deliver comparable results.

4.6 Measuring the similarity of declarative process models

As mentioned at the beginning of Sect. 4, we want to measure the similarity of two non-equal Declare process models. There are two general approaches: (i) considering the automatons as graphs and applying metrics from graph theory and (ii) comparing the automatons on word level. Since in our research domain the second strategy is still neglected, we fill this gap by offering a couple of measurements that are based on the length of traces (Sect. 4.6.1). In Sect. 4.6.2, we propose an additional measure, the *Damerau–Levenshtein distance*, which does not focus on the trace lengths but on the structure of the traces, i.e., regarding the traces as strings and computing their edit distances.

4.6.1 Density and similarity based on trace length

In automata theory, the length of words is not limited at all. However, in business process management we only consider traces of limited length because the number of steps or activities executed for a process is of limited size. Hence, the following measures for comparing process models are based on trace length. Therefore, we first note that for a trace of length n over m activities, there are m^n possible traces. Now we can define the *n-density* of a process model:

Definition 11 For a Declare process model M over m activities, we call

$$\lambda_n(M) := \frac{|\{\sigma \text{ of length } n \mid \sigma \text{ satisfies } M\}|}{m^n} \in [0, 1]$$

the n-**density** of M.

As $\{\sigma$ of length $n \mid \sigma$ satisfies $M\}$ is a subset of all traces of length n, $\lambda_n(M)$ takes a value between 0 and 1. In other words, $\lambda_n(M)$ describes the percentage of traces of length n which satisfies a process model M compared to all potential traces. This measure yields an estimation of how many process traces (of a certain length n) are covered by a process model. The bigger this number, the more flexible a process model is; vice versa, the smaller this number, the more restrictive a process model is. Therefore, the density measure puts the coverage of a process model into perspective.

For two Declare process models M_1, M_2 and $n \in \mathbb{N}$, the corresponding n-densities $\lambda_n(M_1)$ and $\lambda_n(M_2)$ can be calculated by simulating all traces up to length n and checking whether they satisfy M_1, M_2 or none of them. The elements of the respective sets then are counted and hence determine the n-densities. These values can also be used to get a rough feeling about how far the models differ from each other: if the values differ extremely, e.g., $\lambda_n(M_1) = 0.1$ and $\lambda_n(M_2) = 0.7$, M_1 and M_2 cannot have a lot of properties in common (they overlap in at most 10% of the traces and differ in 60%).

Note that a similar n-density does not necessarily mean that the models are similar. Even in the case $\lambda_n(M_1) = 0.5 = \lambda_n(M_2)$, it could be possible that the sets of traces covered by the two models are completely disjoint. Figure 9 depicts all possible cases. The above row shows the case when the sum of the single coverage of the two process models together is not more than 100%. Thus, the two process models can be completely disjoint, can be overlapping or one model can fully encompass the other one. The lower row of Fig. 9 shows the case when two process models together cover more than "100% of process traces", i.e., we again just sum up the coverage of the two process models and so can get a value greater than 100%. Then, the two models might overlap or one might completely encompass the other one. They cannot be disjoint anymore.

We can extend the definition of the n-density to the **min-max-density**, which considers all traces with length between min and max. The explanatory power of this measure is similar to n-density; however, it broadens the scope of observation to a range of trace lengths. The principle proposition of this measure is the same as for the n-density: it unveils the flexibility of a process model.

Definition 12 For a Declare process model M over m activities we call

$$\lambda_{\min}^{\max}(M) := \frac{\sum_{i=\min}^{\max} |\{\sigma \text{ of length } i \mid \sigma \text{ satisfies } M\}|}{\sum_{i=\min}^{\max} m^i} \in [0, 1]$$

the **min-max-density** of M.

Another measure to compare two process models directly is the *n-similarity*:

Definition 13 For two Declare process models M_1 and M_2 over the same (finite) set of activities, we call

$$\Lambda_n(M_1, M_2) := \min_{j \in \{1,2\}} \left\{ \frac{|\{\sigma \text{ of length } n \mid \sigma \text{ satisfies } M_1 \text{ and } M_2\}|}{|\{\sigma \text{ of length } n \mid \sigma \text{ satisfies } M_j\}|} \right\} \in [0, 1]$$

the n-**similarity** of M_1 and M_2.

The main difference of similarity measures to density measures is that the latter compares the coverage of a process model to the whole space of potential process traces. The former takes into account the percentage of traces which are accepted by both models and compares it with the coverage of these models. Consider for example that M_1 accepts 100 traces of length n, M_2 accepts 200 traces of length n and the set of traces of length n which both models accept consists of 50 traces. Then $\Lambda_n(M_1, M_2) = \min\{\frac{50}{100}, \frac{50}{200}\} = \min\{0.5, 0.25\} = 0.25$. This means that 25% of the traces of the "larger" model (i.e., the model which accepts more traces) are accepted by both models, whereas 50% of the more restrictive process model are covered by both models.

The measure n-similarity provides insights on the overlapping of process models to be compared. The bigger this number is, the more the two models are overlapping. A measure of 0 means that the two models are disjoint; a measure of 1 means that they are equal. All numbers between 0 and 1 depict the percentage of common process traces relatively to the less restrictive process model.

Analogously to the min–max density, we define the *min–max similarity*, which describes similarity of two process models with regard to a range of trace lengths. It just broadens the scope of comparison to a range of process traces.

Definition 14 For two Declare process models M_1 and M_2 over the same (finite) set of activities, we call

$$\Lambda_{\min}^{\max}(M_1, M_2) := \min_{j \in \{1,2\}} \left\{ \frac{\sum_{i=\min}^{\max} |\{\sigma \text{ of length } i \mid \sigma \text{ satisfies } M_1 \text{ and } M_2\}|}{\sum_{i=\min}^{\max} |\{\sigma \text{ of length } n \mid \sigma \text{ satisfies } M_j\}|} \right\} \in [0, 1]$$

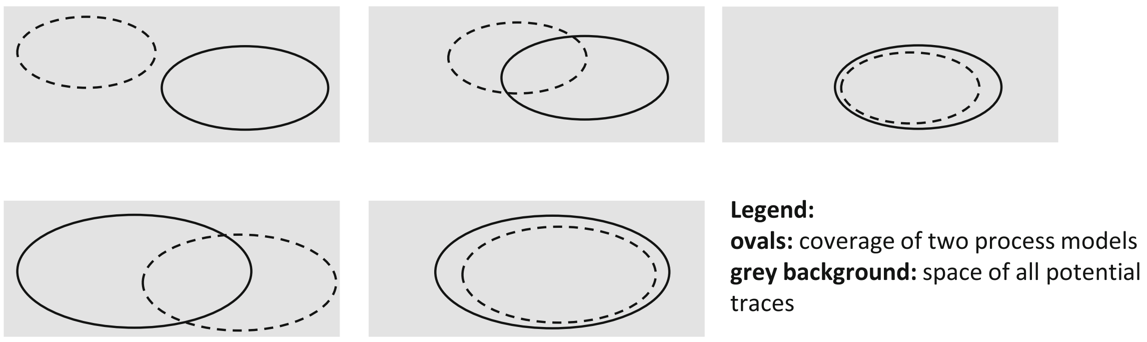

Legend:
ovals: coverage of two process models
grey background: space of all potential traces

Fig. 9 Illustration of coverage for process models

the **min-max-similarity** of M_1 and M_2.

Altogether the two measures density and similarity provide an estimation of how flexible the process models are and how much they overlap. This information helps a process modeler to assess whether to apply one or the other process model. For example, when both process models have about the same density (coverage) and the similarity is pretty high, (s)he "arbitrarily" chooses one of the two process models to be employed, i.e., (s)he chooses that model that looks more familiar or clearer.

4.6.2 Similarity based on Damerau–Levenshtein distance

While in the last subsection, we focus on trace lengths, we now analyze the structure of traces in order to discuss similarity of process models. Comparing or determining the similarity of traces can be done by using metrics that calculate the edit distance between them, i.e., they count the (minimal) number of operations like insertions, deletions, substitutions and transpositions that are needed to transform one trace into another. In context of process traces, however, it is important to take into account that some activities in the process may occur in parallel. Assume, for instance, that in a trace $\sigma_1 = \langle A, C, D, B\rangle$ the activities C and D were executed in parallel (i.e., potentially at the same time). Then, σ_1 could also be rewritten as $\langle A, D, C, B\rangle$. This transposed trace does—in principle—represent "the same" execution. Hence, we use the *Damerau–Levenshtein distance* metric [34], since it does not penalize transpositions so harshly than other metrics based on edit distance. However, in general a transposition cannot be free of any penalization since in many cases the order of execution is crucial. So the Damerau–Levenshtein distance is a good compromise to take into account parallel executions but also not to neglect violations of an execution order.

Definition 15 Let σ_1 and σ_2 be two traces, $i \in \mathbb{N}_{\leq|\sigma_1|}$, $j \in \mathbb{N}_{\leq|\sigma_2|}$ and d a recursive function defined as follows:

$$d_{\sigma_1,\sigma_2}(i,j) := \min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{\sigma_1,\sigma_2}(i-1,j)+1 & \text{if } i > 0 \\ d_{\sigma_1,\sigma_2}(i,j-1) & \text{if } j > 0 \\ d_{\sigma_1,\sigma_2}(i-1,j-1)+\mathbf{1}_{\sigma_1(i)\neq\sigma_2(j)} & \text{if } i,j > 0 \\ d_{\sigma_1,\sigma_2}(i-2,j-2)+1 & \text{if } i,j > 1 \wedge \sigma_1(i) \\ & \neq \sigma_2(j-1) \wedge \sigma_1(i-1) \\ & = \sigma_2(j) \end{cases},$$

where $\mathbf{1}$ denotes the indicator function ($\mathbf{1}_x$ takes the value 1, if x is true, otherwise 0). We call $Lev(\sigma_1, \sigma_2) = d_{\sigma_1,\sigma_2}(|\sigma_1|, |\sigma_2|)$ the **Damerau–Levenshtein distance between** σ_1 and σ_2.

We use the inverse of the scaled measure (for scaling we use the maximum size between the two traces that are compared), so that a higher value implies a higher similarity among the traces, i.e.

$$Lev_{inv}(\sigma_1, \sigma_2) := 1 - \frac{\mathrm{Lev}(\sigma_1, \sigma_2)}{\max(|\sigma_1|, |\sigma_2|)}$$

For determining the similarity between the process models based on a set of traces, we first generate for each process model all traces of a length within an a-priori defined range $[n; m]$. In the following, we denote the set of such traces p_i, i.e., with length $\geq n$ and $\leq m$ of a process model P, as $\mathcal{S}_P^{n,m} = \{p_1, \ldots, p_{|\mathcal{S}_P^{n,m}|}\}$. This set can be considered as a process event log.

Afterward, we pair each generated trace $\sigma \in \mathcal{S}_{M_1}^{n,m}$ of the process model M_1 with the most similar trace (with respect to the Damerau–Levenshtein distance) $\mu \in \mathcal{S}_{M_2}^{n,m}$ of the traces of the process model M_2 [35]. Once the pairs are formed, we calculate the mean Damerau–Levenshtein distance between them [34]:

Definition 16 Let $n, m \in \mathbb{N}_{\geq 0}$ and Lev_{inv} be the inverse and scaled Damerau–Levenshtein distance. For two Declare process models M_1 and M_2, we call

$$\Gamma_n^m(M_1, M_2) := \frac{\sum_{i=1}^{|\mathcal{S}_{M_1}^{n,m}|} \max\{\mathrm{Lev}_{inv}(p_i, k) | k \in \mathcal{S}_{M_2}^{n,m}\}}{|\mathcal{S}_{M_1}^{n,m}|}$$

the m-n-**process event-log-similarity** of M_1 and M_2.

That means that we measure the similarity of all traces with a particular length between two process models. It is important to mention that the min–max-event-log-similarity is in general not symmetric, i.e., $\Gamma_n^m(M_1, M_2) \neq \Gamma_n^m(M_2, M_1)$. This fact can be interpreted as follows: The effort to transform the process model M_1 into M_2 may differ from the effort to transform M_2 into M_1. Depending on the aim of analysis, we calculate $\Gamma_n^m(M_1, M_2)$, $\Gamma_n^m(M_2, M_1)$ or both.

The Damerau–Levenshtein distance directly exhibits on the level of text strings, i.e., traces, how similar two process models are. As with the various measures about density introduced in Sect. 4.6.1, the Damerau–Levenshtein distance is more an indicator than a concrete marker for a statement about similarity. It provides an attested impression of how similar two process models are by comparing traces of these models. Similarity on this level cannot automatically lead to statements about similarity on a conceptual and logical level since small differences on the trace level can lead to great differences on a logical level, and vice versa. However, a domain expert has to assess whether these similarities show the same tendency or are just accidentally similar. We recommend to apply this measure also as an indicator for potential similarities of process models.

4.6.3 Example

We now apply the previously defined measures to the process models Q_1 and Q_2 of the running example (Sect. 2.4). All values are depicted in Table 2.

For $n = 0$, there is only the empty trace which is accepted neither by Q_1 nor by Q_2. That is why the densities of both process models are 0 for $n = 0$. Nevertheless, we count the empty trace as one potential trace. Thus, there is a 1 in the denominator of the fraction in the upper four rows in the column for $n = 0$. As Q_2 requires the execution of minimal two Bs, it does not accept a trace of length 1 and only accepts one trace of length 2, namely $\langle BB \rangle$. This restriction causes that the density of Q_1 is larger than the density of Q_2 for $n = 1, 2$. For $n = 3$, we see that Q_1 and Q_2 accept the same number of traces and hence have the same density λ_n. We observe that for $n \geq 4$ the values of the density measures (λ_n, λ_0^n) for Q_1 are lower than for Q_2. This can be interpreted that there is a significant difference between the process models and that process model Q_2 describes a more flexible process, since it offers more process execution variants, which is confirmed by Fig. 10: the difference between these measures of Q_1 and Q_2 increases for greater n.

Table 2 Measures for process models Q_1 and Q_2 of the running example

n	0	1	2	3	4	5	6	7
$\lambda_n(Q_1)$	$\frac{0}{1}$	$\frac{1}{3}$	$\frac{3}{9}$	$\frac{7}{27}$	$\frac{15}{81}$	$\frac{31}{243}$	$\frac{63}{729}$	$\frac{127}{2187}$
$\lambda_n(Q_2)$	$\frac{0}{1}$	$\frac{0}{3}$	$\frac{1}{9}$	$\frac{7}{27}$	$\frac{33}{81}$	$\frac{131}{243}$	$\frac{473}{729}$	$\frac{1611}{2187}$
$\lambda_0^n(Q_1)$	$\frac{0}{1}$	$\frac{1}{4}$	$\frac{4}{13}$	$\frac{11}{40}$	$\frac{26}{121}$	$\frac{57}{364}$	$\frac{120}{1093}$	$\frac{247}{3280}$
$\lambda_0^n(Q_2)$	$\frac{0}{1}$	$\frac{0}{4}$	$\frac{1}{13}$	$\frac{8}{40}$	$\frac{41}{121}$	$\frac{172}{364}$	$\frac{645}{1093}$	$\frac{2256}{3280}$
$\Lambda_n(Q_1, Q_2)$	0	0	0	0	0	0	0	0
$\Lambda_0^n(Q_1, Q_2)$	0	0	0	0	0	0	0	0
$\Gamma_0^n(Q_1, Q_2)$	–	–	0	0.33	0.5	0.6	0.67	0.71
$\Gamma_0^n(Q_2, Q_1)$	–	–	0	0.29	0.43	0.51	0.56	0.6

The similarity measures $\Lambda_n(Q_1, Q_2)$ and $\Lambda_0^n(Q_1, Q_2)$ are 0 for all $n \in \mathbb{N}$, which implies that Q_1 and Q_2 do not accept common traces. This is caused by the fact that the existence$(A, 1)$ and exclusiveChoice(A, B) constraints of Q_1 prohibit the execution of activity B and the existence$(B, 2)$ implies an—at least—double execution of activity B. Hence, activity B may not occur in the execution of Q_1, whereas the execution of Q_2 requires the execution of B. Taking the observations regarding densities and similarities together, we can conclude that we are in the upper left case of Fig. 9, i.e., the two processes are disjoint—with respect to execution traces—and differ broadly with respect to their flexibility.

Table 2 also reveals that the measure 0-n-process event-log-similarity is indeed asymmetric. The graphical curves depicted in Fig. 10 confirm this observation. We can also observe that with increasing trace length the similarity values of the 0-n-process event-log-similarity increase and for all $n \geq 3$ $\Gamma_0^n(Q_1, Q_2) > \Gamma_0^n(Q_2, Q_1)$ holds. This is again caused by the two constraints existence$(A, 1)$ and exclusiveChoice(A, B) of Q_1 which prohibit the execution of activity B. Hence, from each accepted trace t_1 of Q_1 an accepted trace t_2 of Q_2 can be derived by replacing any two symbols of t_1 by Bs (as two Bs are mandatory for Q_2). The other way around, for transforming a trace t_2 of Q_2 into a trace t_1 of Q_1, all Bs can be exchanged by As. Hence, a trace t_1 of Q_1 can be derived from a trace t_2 of Q_2 by replacing each B by A. As t_2 includes at least two Bs, at least two operations are needed. As some traces of Q_2 include more than two Bs, more than two operations are needed, which implies a higher inverse Damerau–Levenshtein distance and hence a lower 0-n-process event-log-similarity.

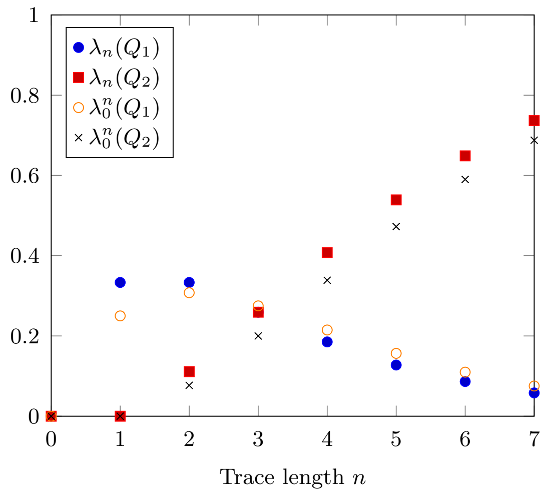


Fig. 10 Measure curves for Q_1 and Q_2 of the running example

5 Implementation and evaluation

In this section, we give an introduction to our implementation and evaluate our approach from two angles: First, we determine the time complexity of our approach (cf. Sect. 5.1). Analyzing the asymptotic behavior has the advantage that the results are independent of the deployed hardware and more general than calculating particular example processes. Afterward, we conduct a small comparative study of declarative mining approaches on real-life event logs to demonstrate the applicability of our approach in a practical scenario (cf. Sect. 5.3).

5.1 Implementation

As a proof of concept, the algorithm visualized in Fig. 4 has been implemented in Java and can be used through a command-line interface. All steps are arranged in a configurable and automated pipeline. Besides the sources and a pre-compiled runnable JAR file also some sample models are publicly available.[4]

As mandatory inputs, the user has to provide two models as text files of the following structure:

$A, B, C, D; precedence(A, B); init(C) \dots .$

Activities have to be encoded in the shape of a comma-separated list of characters—the alphabet—at the beginning of each text file, followed by an arbitrary number of Declare constraints in the commonly used textual representation: *< constraintTemplateName >*(*< activity1 >*[,*< activity2 >*]). Activities need to be part of the alphabet.

In order to run the application, providing the model files is sufficient. However, it is possible to output the automatons derived from the Declare models using the parameter *--dfa-output < path >*. With *–max-word-length < number >*, it is possible to configure the maximum trace length used for computing similarity measures for the given models by comparing the traces that are conform to each model respectively (cf. Sect. 4.6.1). By omitting this parameter, a default value is used.

5.2 Time complexity analysis

We discuss the time complexity of both the simulation-based and the theory-based approach. They both have the first step of constructing the minimized product automaton from the constraints of a process model in common. For the minimization, we use the Hopcroft algorithm [14], that has a time complexity of $\mathcal{O}(n \log \log n)$ where n denotes the number of states of the automaton.

In the following, we regard two process models $M_1 = (\mathcal{A}_1, \mathcal{T}_1)$ and $M_2 = (\mathcal{A}_2, \mathcal{T}_2)$, where $\mathcal{T}_1 = \{t_1, \dots, t_n\}$ and $\mathcal{T}_2 = \{t_1^{'}, \dots, t_m^{'}\}$. For a template $t \in \mathcal{T}$, we denote the number of states of the corresponding template automaton by $|S_t|$. Hence, the minimal product automatons of M_1 and M_2 can be calculated in:

$$\begin{aligned}
R{:=}\mathcal{O}\Bigg(&\sum_{i=1}^{n-1}\Big(\mathcal{O}\left(|S_{t_i}| \cdot |S_{t_{i+1}}|\right)\\
&+\mathcal{O}\left(|S_{t_i}| \cdot |S_{t_{i+1}}| \cdot \log\log\left(|S_{t_i}| \cdot |S_{t_{i+1}}|\right)\right)\Big)\Bigg)\\
&+\mathcal{O}\left(\sum_{i=1}^{m-1}\left(\mathcal{O}\left(|S_{t_i^{'}}| \cdot |S_{t_{i+1}^{'}}|\right)\right.\right.\\
&\left.\left.+\mathcal{O}\left(|S_{t_i^{'}}| \cdot |S_{t_{i+1}^{'}}| \cdot \log\log\left(|S_{t_i^{'}}| \cdot |S_{t_{i+1}^{'}}|\right)\right)\right)\right)
\end{aligned}$$

Simulation-based The most computational intensive task is the generation and checking of the traces. In dependency of the applied technique (i.e., SAT solving), the time complexity differs. We denote the time complexity in dependency of the considered process model $\mathcal{M}$ and the maximal trace length n with $\gamma(n, \mathcal{M})$. SAT solving for propositional logic is known to be NP-complete (Cook–Levin theorem [36]). Hence, the time complexity for generating and validating traces is exponential and also dominates the time complexity of the simulation-based algorithm. Note that in this case determining the time complexity does not primarily evaluate the algorithm itself rather than the applied SAT solver. Hence, we have in summary the following asymptotic behavior for the simulation-based algorithm, where the first term describes the time complexity of constructing the corresponding minimal product automatons. The second and the third terms describe the time complexity for generating and

[4] https://github.com/mkaep/comparing-declare-models.

Table 3 Statistic of the used event logs

	Helpdesk	BPIC12	BPIC13
#Traces	4580	13,087	1487
#Events	21,348	262,200	6660
#Activities	14	24	4
AVG trace length	4.66	20.04	4.48
Max. trace length	15	175	35
Min trace length	2	3	1
#Trace variants	226	4366	183

checking the traces until the upper bound b:

$$\mathcal{O}\left(R + \gamma(b, \mathcal{M}_1) + \gamma(b, \mathcal{M}_2)\right).$$

Since the last terms are the predominately ones, the time complexity of the simulation-based algorithm is exponential. However, if we are only interested in the minimal upper bound, the time complexity is R.
Theory-based For two finite state automatons, the theory-based algorithm has time complexity $R + \mathcal{O}(m \cdot n)$ where m and n are the number of states of the minimal product automatons of M_1 and M_2. The first term describes again the construction of the minimal product automatons and the second term describes the time complexity of the symmetric difference construction. We observe that the theory-based algorithm is much faster than the simulation-based algorithm as the dominating term $m \cdot n$ is quadratic and not exponential. *Analyzing Differences* If the two process models are not equal and we are interested in analyzing the differences (cf. Sect. 4.6), it is necessary to calculate all traces up to a desired length l. This task requires as mentioned above an exponential time complexity of $\gamma(l, \mathcal{M}_1) + \gamma(l, \mathcal{M}_2)$.

5.3 Practical application

For evaluating how our approach performs on real-life data, we conduct a small comparative study of declarative mining algorithms. We apply two Declare Miner (UnconstrainedMiner [28] and DeclareMapsMiner [37]) to extract declarative process models from real-life event logs. Afterward, we use our approach (and metrics) for a comparison of the mined models. We performed our study on 3 real-life event logs from different domains with diverse characteristics (cf. Table 3) extracted from the *4TU Center for Research Data*.[5]

We configured both Declare Miner in that way, that all supported Declare templates (cf. Table 1) of our approach were mined. Additionally, we set the threshold for confidence and support that a constraint must satisfy to ≥ 0.9. Setting the

Table 4 Measures of the λ_n metric

Event log	Miner	0	1	2	3	4	5
Helpdesk	MapsMiner	$\frac{0}{1}$	$\frac{0}{14}$	$\frac{0}{14^2}$	$\frac{0}{14^3}$	$\frac{1}{14^4}$	$\frac{2}{14^5}$
Helpdesk	UncMiner	$\frac{1}{1}$	$\frac{1}{14}$	$\frac{1}{14^2}$	$\frac{0}{14^3}$	$\frac{1}{14^4}$	$\frac{7}{14^5}$
BPIC13	MapsMiner	$\frac{0}{1}$	$\frac{0}{4}$	$\frac{1}{4^2}$	$\frac{1}{4^3}$	$\frac{3}{4^4}$	$\frac{6}{4^5}$
BPIC13	UncMiner	$\frac{1}{1}$	$\frac{0}{4}$	$\frac{1}{4^2}$	$\frac{1}{4^3}$	$\frac{3}{4^4}$	$\frac{6}{4^5}$
BPIC12	MapsMiner	$\frac{0}{1}$	$\frac{0}{24}$	$\frac{1}{24^2}$	$\frac{19}{24^3}$	$\frac{362}{24^4}$	$\frac{6860}{24^5}$
BPIC12	UncMiner	$\frac{1}{1}$	$\frac{5}{24}$	$\frac{80}{24^2}$	$\frac{3654}{24^3}$	$\frac{81,923}{24^4}$	$\frac{890,237}{24^5}$

Table 5 Measures of the λ_0^n metric

Event log	Miner	0	1	2	3	4	5
Helpdesk	MapsMiner	$\frac{0}{1}$	$\frac{0}{15}$	$\frac{0}{211}$	$\frac{0}{2955}$	$\frac{1}{41,371}$	$\frac{3}{579,195}$
Helpdesk	UncMiner	$\frac{0}{1}$	$\frac{1}{15}$	$\frac{2}{211}$	$\frac{2}{2955}$	$\frac{3}{41,371}$	$\frac{10}{579,195}$
BPIC13	MapsMiner	$\frac{1}{1}$	$\frac{1}{5}$	$\frac{2}{21}$	$\frac{3}{85}$	$\frac{6}{341}$	$\frac{12}{1365}$
BPIC13	UncMiner	$\frac{0}{1}$	$\frac{0}{5}$	$\frac{1}{21}$	$\frac{2}{85}$	$\frac{5}{341}$	$\frac{11}{1365}$
BPIC12	MapsMiner	$\frac{0}{1}$	$\frac{0}{25}$	$\frac{1}{601}$	$\frac{20}{14,425}$	$\frac{382}{346,201}$	$\frac{7242}{8,308,825}$
BPIC12	UncMiner	$\frac{1}{1}$	$\frac{6}{25}$	$\frac{86}{601}$	$\frac{3740}{14425}$	$\frac{85,663}{346,201}$	$\frac{975,900}{8,308,825}$

Table 6 Measures of the $\Gamma_0^n(UncMiner, MapsMiner)$ metric

Event log	0	1	2	3	4	5
Helpdesk	0	0	1	1	$\frac{2}{3}$	$\frac{3}{6}$
BPIC13	0	0	0	1	1	$\frac{2}{7}$
BPIC12	0	0	0	0.3	0.4	0.3

support lower than 1.0 is necessary, since all real-life event logs contain noise. On the other hand, a smaller threshold leads to a significant increase of constraints (several thousand constraints), which resulted for all used event logs in a broken model, i.e., the model was so restrictive that it prevents any process execution.

We could directly observe that the mined process models differ with regard to the number of constraints (cf. Table 7). In all cases, the UnconstrainedMiner detects a larger number of constraints. Hence, the UnconstrainedMiner bears a higher risk for a broken model. In all cases, the models were different and no one was included in the other one. So we calculated the metrics proposed in the previous section to quantify the difference between the models. The measurements are listed in Tables 4, 5 and 6. Since the metrics require a simulation of all traces up to a given length, the calculation of the metrics is faced with the same problem as of any simulation-based approach. The scalability wall prevents a (fast) simulation of long traces. Hence, we set an upper bound of 5 for calculating the simulation based metrics. We argue that this limit is already sufficient, since larger traces would be very cumbersome for a manual investigation by a domain expert and even these short traces allow to derive a tendency. Note that this scalability problem only holds for the calculation of the simulation based metrics, while the equality check based on

[5] https://data.4tu.nl/.

Table 7 Statistics of the mined process models

Event log	#Constraints MapsMiner	#Constraints UnconstrainedMiner
Helpdesk	94	363
BPIC13	9	34
BPIC12	13	67

automatons is not affected. Hence, we can conclude that our approach enables the comparison of large declarative models as they occur on real data. The analysis of the outputted traces up to length 5 reveals that the models are not broken, but even very restrictive, due to the small amount of allowed process executions. Nevertheless, even these small number provide useful insights for a domain expert. In all cases, we could observe that albeit neither the process models are identical nor one is a subset of the other, the intersection of allowed traces between the models was not empty. Hence, the mined models possess similarities and allow in partial the same behavior. We could also derive from the measures n-density and 0-n-density that the models of the MapsMiner are more restrictive. Eventually it could also determined that the UnconstrainedMiner allows in all cases an empty trace, while the MapsMiner prevents this behavior.

6 Conclusion and future work

In this paper, we presented two different approaches for comparing two Declare process models for equality by using finite state automaton constructions and minimization. The first approach, the simulation-based approach, makes use of a calculated upper bound, which was proved. The corresponding algorithm shows an exponential time complexity, whereas the second approach, the theory-based algorithm, performs quadratic and hence surpasses the simulation-based approach. On the other side, the simulation-based approach is needed in order to make statements about common properties and differences of models, which are not completely identical.

In future work, our approach will be extended to other process modeling languages, especially declarative multi-perspective languages like MP-Declare [4]. Whereas Declare mainly considers the control flow perspective [10], MP-Declare can also deal with human and technical resources like performing actors or used artifacts (e.g., computer programs, tools). Furthermore, we aim to make our approach applicable for so-called imperative process modeling languages like the Business Process Model and Notation (BPMN) [38] in order to construct a tool, which can compare a plethora of different process modeling languages. Finally, the approach will be integrated in a graphical user friendly interface.

Another important point in our future work will be to elaborate and discuss the applicability of our approach to other domains, e.g. organizational models or sequence diagrams. As organizational models are "static" in a way, there might be no direct application of our approach, whereas investigating sequence diagrams might lead to promising results as there are already efforts of transforming them into finite state automatons [39].

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Fahland, D., Mendling, J., Reijers, H.A., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: the issue of maintainability. In: Business Process Management Workshops, pp. 477–488. Springer (2010)
2. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: the issue of understandability. Springer (2009)
3. Pesic, M.: Constraint-based workflow management systems: shifting control to users. Ph.D. thesis, Industrial Engineering and Innovation Sciences (2008)
4. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)
5. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Log. Algebraic Program. **82**(5–7), 164–185 (2013)
6. Zeising, M., Schönig, S., Jablonski, S.: Towards a common platform for the support of routine and agile business processes. In: Collaborative Computing: Networking, Applications and Worksharing (2014)
7. Schönig, S., Ackermann, L., Jablonski, S.: Towards an implementation of data and resource patterns in constraint-based process models. In: Modelsward (2018)
8. Abbad Andaloussi, A., Burattin, A., Slaats, T., Petersen, A.C., Hildebrandt, T.T., Weber, B.: Exploring the Understandability of a Hybrid Process Design Artifact Based on DCR Graphs, pp. 69–84. Springer, Cham (2019)

9. Schützenmeier, N., Käppel, M., Petter, S., Jablonski, S.: Upper-bounded model checking for declarative process models. In: Serral, E., Stirna, J., Ralyté, J., Grabis, J. (eds.) The Practice of Enterprise Modeling, pp. 195–211. Springer, Cham (2021)
10. van de Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Wiesbaden (2011)
11. Zuck, L.: Past temporal logic. Ph.D. thesis, Weizmann Institute, Israel (1986)
12. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science (2002)
13. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Pearson/Addison Wesley, Boston (2007)
14. Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press, Cambridge (1971). https://doi.org/10.1016/B978-0-12-417750-5.50022-1
15. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.: Process equivalence: comparing two process models based on observed behavior. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) Business Process Management, pp. 129–144. Springer, Berlin (2006)
16. Becker, M., Laue, R.: A comparative survey of business process similarity measures. Comput. Ind. **63**(2), 148–167 (2012). https://doi.org/10.1016/j.compind.2011.11.003. (**Managing Large Collections of Business Process Models**)
17. La Rosa, M., Dumas, M., Ekanayake, C., García-Bañuelos, L., Recker, J., Hofstede, A.: Detecting approximate clones in business process model repositories. Inf. Syst. **49**, 102–125 (2015). https://doi.org/10.1016/j.is.2014.11.010
18. Tealeb, A., Awad, A., Galal-Edeen, G.: Context-based variant generation of business process models, vol. 175, pp. 363–377 (2014)
19. Haisjackl, C., Barba, I., Zugal, S., Soffer, P., Hadar, I., Reichert, M., Pinggera, J., Weber, B.: Understanding declare models: strategies, pitfalls, empirical results. Softw. Syst. Model. **15**(2), 325–352 (2016)
20. Andaloussi, A.A., Buch-Lorentsen, J., Lopez, H.A., Slaats, T., Weber, B.: Exploring the modeling of declarative processes using a hybrid approach. In: Proceedings of 38th International Conference on Conceptual Modeling 2019, pp. 162–170. Springer (2019)
21. Ciccio, C.D., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017)
22. Smedt, J.D., Weerdt, J.D., Serral, E., Vanthienen, J.: Discovering hidden dependencies in constraint-based declarative process models for improving understandability. Inf. Syst. **74**(Part), 40–52 (2018)
23. De Smedt, J., De Weerdt, J., Serral, E., Vanthienen, J.: Improving understandability of declarative process models by revealing hidden dependencies. In: Advanced Information Systems Engineering, pp. 83–98. Springer (2016)
24. Schützenmeier, N., Käppel, M., Petter, S., Schönig, S., Jablonski, S.: Detection of declarative process constraints in LTL formulas. In: EOMAS-15th International Workshop 2019, Selected Papers, LNBIP, vol. 366, pp. 131–145. Springer (2019)
25. Dijkman, R., Dumas, M., García-Bañuelos, L., Käärik, R.: Aligning business process models, pp. 45–53 (2009). https://doi.org/10.1109/EDOC.2009.11
26. Shi, Y., Xiao, S., Li, J., Guo, J., Pu, G.: Sat-based automata construction for LTL over finite traces. In: 27th Asia-Pacific Software Engineering Conference (APSEC) (2020)
27. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. Springer, Berlin (2001)
28. Westergaard, M., Stahl, C., Reijers, H.: UnconstrainedMiner: efficient discovery of generalized declarative process models. BPM reports. BPMcenter.org (2013)
29. Corea, C., Nagel, S., Mendling, J., Delfmann, P.: Interactive and minimal repair of declarative process models, pp. 3–19 (2021). https://doi.org/10.1007/978-3-030-85440-9_1
30. Hidders, J., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Verelst, J.: When are two workflows the same? In: Proceedings of the 2005 Australasian Symposium on Theory of Computing-Volume 41, CATS '05, pp. 3–11. Australian Computer Society, Inc., Sydney (2005)
31. Käppel, M., Schönig, S., Ackermann, L., Jablonski, S.: Language-independent look-ahead for checking multi-perspective declarative process models. Softw. Syst. Model. **20**, 1379–1401 (2021)
32. Ackermann, L., Schönig, S., Petter, S., Schützenmeier, N., Jablonski, S.: Execution of multi-perspective declarative process models. In: OTM 2018 Conferences (2018)
33. Skydanienko, V., Francescomarino, C.D., Maggi, F.: A tool for generating event logs from multi-perspective declare models. In: BPM (Demos) (2018)
34. Boytsov, L.: Indexing methods for approximate dictionary searching: comparative analysis. ACM J. Exp. Algorithmics **16**, 1 (2011). https://doi.org/10.1145/1963190.1963191
35. Camargo, M., Dumas, M., González-Rojas, O.: Learning accurate LSTM models of business processes. In: Hildebrandt, T., van Dongen, B.F., Röglinger, M., Mendling, J. (eds.) Business Process Management, pp. 286–302. Springer, Cham (2019)
36. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71. ACM, New York (1971)
37. Maggi, F.M.: Declarative process mining with the declare component of prom. In: Fauvet, M., van Dongen, B.F. (eds.) Proceedings of the BPM Demo sessions 2013, Beijing, China, August 26–30, 2013, CEUR Workshop Proceedings, vol. 1021. CEUR-WS.org (2013)
38. OMG: Business Process Model and Notation (BPMN), Version 2.0. http://www.omg.org/spec/BPMN/2.0 (2011)
39. Duan, Z., Yu, B., Zhang, C., Tian, C., Ding, M.: A test case generation approach based on sequence diagram and automata models. Chin. J. Electron. **25**, 234–240 (2016). https://doi.org/10.1049/cje.2016.03.007

Nicolai Schützenmeier is a research assistant with the Institute for Computer Science at University of Bayreuth (Germany). He received the master's degree at University of Bayreuth. His research is focused on BPM, especially declarative process models and their mathematically and theoretical foundations. Based on his work, he published scientific papers in this area in international conferences.

Martin Käppel is a research assistant with the Institute for Computer Science at University of Bayreuth (Germany). He received the master's degree (with honours) at University of Bayreuth. His research is focused on Process Mining, especially predictive business process monitoring and the development of Small Sample Learning methods for BPM. Based on his work, he published scientific papers in international conferences and journals.

Lars Ackermann is an Assistant Professor of Computer Science with the Institute for Computer Science at University of Bayreuth (Germany). He received the master's degree (with honours) in Computer Science and the doctoral degree from University of Bayreuth. He has an established background in BPM/ Process Mining and has been working in this field for several years. He published extensively in the research area of business process management and information systems, both in international conferences and journals.

Stefan Jablonski is a Full Professor of Computer Science with the Institute for Computer Science at University of Bayreuth (Germany). He is head of the chair for Databases and Information Systems. His major research interests include Business Process Management, flexible process enactment technologies and metamodelling. He has been participating in numerous national and international BPM research as well as industrial projects.

Sebastian Petter is a PhD student at the University of Bayreuth (Germany). He is working as research assistant at the Chair for Databases and Information Systems with a focus on process management and its fusion with recommendation systems. Sebastian holds a master's degree from the University of Bayreuth.