Upper-Bounded Model Checking for Declarative Process Models

Nicolai Schützenmeier, Martin Käppel, Sebastian Petter, Stefan Jablonski

Upper-bounded Model Checking for Declarative Process Models

Nicolai Schützenmeier[1], Martin Käppel[1], Sebastian Petter[1], and Stefan Jablonski[1]

Institute for Computer Science, University of Bayreuth, Germany
{nicolai.schuetzenmeier, martin.kaeppel, sebastian.petter, stefan.jablonski}@uni-bayreuth.de

Abstract. Declarative process modelling languages like Declare focus on describing a process by restrictions over the behaviour, which must be satisified throughout process execution. Although this paradigm allows more flexibility, it has been shown that such models are often hard to read and understand, which affects their modelling, execution and maintenance in a negative way. A larger degree of flexibility leads to a multitude of different process models that describe the same process. Often it is difficult for the modeller to keep the model as simple as possible without over- or underspecification. Hence, model checking, especially comparing declarative process models on equality becomes an important task. In this paper, we determine and prove a theoretical upper bound for the trace length up to which the process executions of Declare models must be compared, to decide with certainty whether two process models are equal or not.

Keywords: Linear Temporal Logic · Model Checking · Declarative Process Management.

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) [10,9]. For the latter, in the last years a couple of different process modelling languages such as Declare [18], Multi-Perspective-Declare (MP-Declare) [5], DCR graphs [13], and the Declarative Process Intermediate Language (DPIL) [25,19] emerged. These languages describe a process by restrictions (so-called *constraints*) over the behaviour, which must be satisfied throughout process execution. Especially Declare has become a widespread and frequently used modelling language in the research area of modelling single-perspective (i.e. focussing on the control-flow) and flexible processes.

Although this paradigm guarantees more flexibility than the imperative one, it turned out that declarative process models are for several reasons hard to read and understand, which affects the execution, modelling, and maintenance of declarative process models in a negative way: the large degree of flexibility

offers the modeller a multitude of options to express the same fact. Hence the same process can be described by very different declarative process models (cf. Section 2). In general, declarative process models possess a high risk for over- or underspecification, i.e. the process model forbids valid process exeuctions or it allows process executions that do not correspond to reality, respectively. Often such a wrong specification is caused by hidden dependencies [2], i.e., implicit dependencies between activitities that are not explicitely modelled but occur through the interaction of other dependencies. The Declare modelling language relies on linear temporal logic (LTL) [18]. 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) modellers 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 unique. Mixing 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 underspecification. Hence model checking, especially comparing models on equality, becomes an important task for modelling and verifying declarative process models. Most of the time this is achieved by simulating process executions of different length (so-called trace length) and by 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.

In this paper we determine and prove a theoretical upper bound for the trace length up to which the process executions must be compared to decide with certainty whether two process models are equal or not.

The rest of the paper is structured as follows: Section 2 recalls basic terminology and introduces a running example. In Section 3 we give an overview of related work and show how our work differs from existing work. In Section 4 we determine an upper bound, prove our claim, discuss existing limitations and the expandability to other declarative process modelling languages. Finally, Section 5 draws conclusions from the work and gives an outlook on future work.

2 Running Example and Basic Terminology

In this section we recall basic terminology and introduce a running example. Events, traces and logs are introduced to provide a common basis for the contents of both process models and process traces. Afterwards we give a short introduction of the Declare modelling language, since we focus on this modelling language in the rest of the paper.

2.1 Events, Traces and Logs

We briefly recall the standard definitions of events, traces and process logs as defined in [1]: An event is an occurence of an activity (i.e. a well-defined step in a business process) in a particular process instance. A trace is a time ordered sequence of events which belongs to the execution of the same process instance. Hence, a trace can be viewed as a record of a process execution. A (process) event log is a multiset of traces. We can now define these terms more formally:

Definition 1. *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 use the notation $\sigma(i)$ to refer to the ith element in σ.*

We say 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 modelling languages like Declare the user must stop working on the process instance to close them, whereas in imperative process models this is achieved automatically by reaching an end event [18]. However, a process instance can only be closed if and only if no constraint of the underlying process model is violated [18].

From the definitions above, we can derive the definition of an event log.

Definition 2. *An **event log** is a multiset $[\sigma_1^{w_1}, ..., \sigma_n^{w_n}]$ of completed traces with $w_i \in \mathbb{N}_+$.*

2.2 Declare and Declare Constraints

Declare is a single-perspective declarative process modelling language that was introduced in [18]. Instead of modelling all viable paths explicitly, Declare describes a set of constraints applied to activities that must be satisfied throughout process execution. Hereby, the control-flow and the ordering of the activities is implicitly specified. All process executions that do not violate a constraint are allowed. In Declare the constraints are instances of templates, i.e. patterns that define parameterized classes of properties [5]. Each template possesses a graphical representation in order to make the model more understandable to the user. Table 1 summarizes the common Declare 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 modeller. 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) [18]. Hence, we can define a Declare process model formally as follows:

Definition 3. *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 (i.e. instances of the predefined templates or LTL formulas).*

Template	**LTL$_f$ Semantics**
existence(A)	$\mathbf{F}(A)$
absence(A)	$\neg\mathbf{F}(A)$
atLeast(A, n)	$\mathbf{F}(A \wedge \mathbf{X}(\text{atLeast}(A, n-1))), \text{atLeast}(A, 1) = \mathbf{F}(A)$
atMost(A, n)	$\mathbf{G}(\neg A \vee \mathbf{X}(\text{atMost}(A, n-1))), \text{atMost}(A, 0) = \mathbf{G}(\neg A)$
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)	$\mathbf{G}(A \rightarrow \mathbf{X}(B)) \wedge \text{response}(A, B)$
precedence(A, B)	$\mathbf{F}(B) \rightarrow ((\neg B)\mathbf{U}A)$
alternatePrecedence(A, B)	$\text{precedence}(A, B) \wedge \mathbf{G}(B \rightarrow \mathbf{X}(\text{precedence}(A, B))$
chainPrecedence(A, B)	$\text{precedence}(A, B) \wedge \mathbf{G}(\mathbf{X}(B) \rightarrow A)$
succession(A, B)	$\text{response}(A, B) \wedge \text{precedence}(A, B)$
chainSuccession(A, B)	$\mathbf{G}(A \leftrightarrow \mathbf{X}(B))$
alternateSuccession(A, B)	$\text{alternateResponse}(A, B) \wedge \text{alternatePrecedence}(A, B)$
notRespondedExistence(A, B)	$\mathbf{F}(A) \rightarrow \mathbf{F}(B)$
notResponse(A, B)	$\mathbf{G}(A \rightarrow \neg\mathbf{F}(B))$
notPrecedence(A, B)	$\mathbf{G}(F(B) \rightarrow \neg A)$
notChainResponse	$\mathbf{G}(A \rightarrow \neg\mathbf{X}(B))$
notChainPrecedence(A, B)	$\mathbf{G}(\mathbf{X}(B) \rightarrow \neg A)$
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)	$(\mathbf{F}(A) \vee \mathbf{F}(B)) \wedge \neg(\mathbf{F}(A) \wedge \mathbf{F}(B))$

Table 1: Semantics for Declare constraints in LTL$_f$

LTL makes statements about the future of a system possible. 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) [26] by introducing so-called past operators, which makes statements on the past possible but does not increase the expressiveness of the formalism [17]. 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

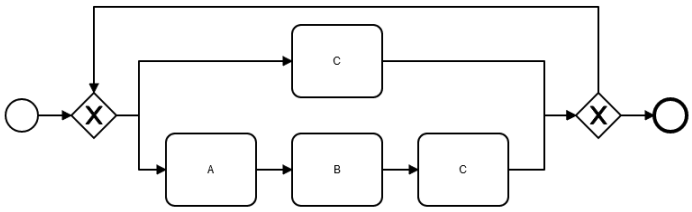


Fig. 1: Running example modelled with BPMN

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 template 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 $t_4 = \langle A, B, A, C \rangle$. In traces t_1, t_2 and t_3 the response template is satisfied. Note that in t_2 this constraint is trivially fulfilled since A does not occur (so-called *vacuously satisfied*). However, t_4 violates the constraint, because after the second occurrence of A no execution of B follows.

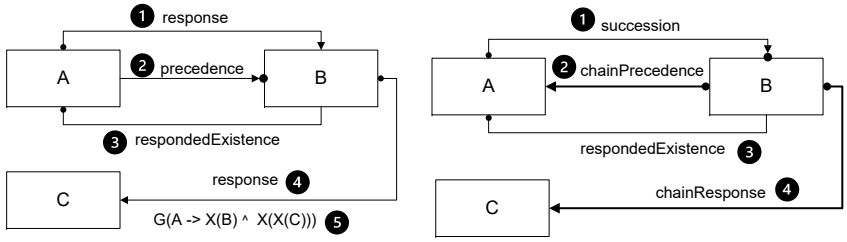
We say that an event activates a constraint in a trace if its occurence imposes some obligations on other events in the same trace. Such an activation either leads to a fulfillment or to a violation of a constraint. Consider, for example, the response template. This constraint is activated by the execution of activity A. In t_4, for instance, the response template is activated twice. In case of the first activation, this leads to a fulfillment, because B occurs. However, the second activations leads to an violation, because B does not occur subsequently.

2.3 Running Example

In this paper we will refer to the following running example. Our sample process consists of three activities namely, 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. In other words, the process can be considered as a loop where in each pass you have the decision to execute activity C or the sequence ABC. For a better understanding, the process model is also shown as BPMN diagram in Fig. 1. This process can be modelled in Declare in different ways. Fig. 2a shows a first option and Fig. 2b shows a second option. For representing the two models we use the common graphical Declare notation.

Process Model M_1 (cf. Fig. 2a):

1. If A is executed,B must be executed sometimes in the future.
2. If B is executed, A must be executed sometimes before.
3. If A occurs, B must also be executed (either before or after A).
4. If B is executed, C must be executed sometimes in the future.
5. If A is executed, B must be executed directly afterwards and C directly after B.



(a) Process Model M_1 (b) Process Model M_2

Fig. 2: Two different Declare process models, describing the same process.

Process Model M_2 (cf. Fig. 2b):

1. If A is executed, B must eventually be executed sometimes in the future and if B is executed, A must be executed sometimes before.
2. If B is executed, A must have been executed directly before.
3. If A occurs, B must also be executed (either before or after A).
4. If B is executed, C must be executed immediately afterwards.

Note that the two process models are made more complicated than necessary in order to demonstrate the difficulty of the comparing task. Apart from the *respondedExistence* template that occurs in both process models, they seem to be completely different with regard to both the applied Declare templates and the number of constraints. Note that in addition to these two variants there are several further options for modelling this process. In the rest of this paper we will explain all steps of this example and we will check whether the two process models are really the same.

3 Related Work

This work relates to the stream of research on modelling and checking declarative process models. Difficulties in understanding and modelling declarative processes are a well-known problem in current research. Nevertheless, there are only a handful of experimental studies that deal with the understandability of declarative process models. In [12] a study reveals that single constraints can be handled well by most individuals, whereas sets of constraints establish a serious challenge. Furthermore, it has been stated that individuals use the composition of the notation elements for interpreting Declare models. Similar studies [4,2] 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 [6] an automata based approach is presented for the detection of redundant constraints and contradictions between the constraints. In [23,8] the problem of the detection of hidden dependencies is addressed. In [23] the extracted hidden dependencies are added to the Declare models through visual and textual annotations to improve the understandability of the models. In [20] the authors transform the

common Declare templates in a standardized form called positive normalform, with the aim of simplifying model comparisons. However, the proposed representation is not sufficient for a reliable identification of identical process models, since this normalform is not unique.

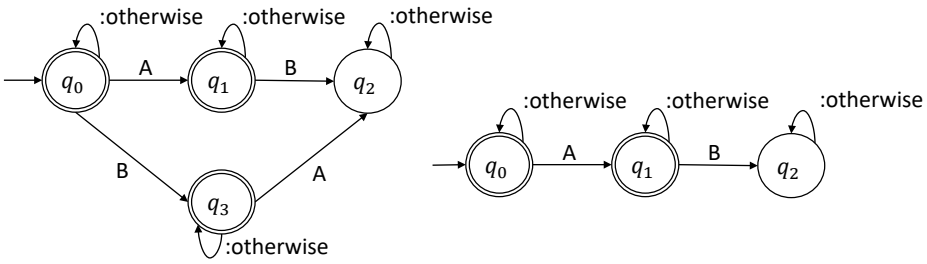
There is also some effort in transforming Declare process models into different representations for deeper analysis. In [21] formulas of linear temporal logic over finite traces are translated to both nondeterministic and deterministic finite automata. In [11] Büchi automata are generated from LTL formulas. In [24] Declare templates are translated into deterministic finite automata, that are used for implementing a declarative discovery algorithm for the Declare language.

The standard procedure for comparing the desired behaviour with the expected behaviour provided in a process model includes the generation of exemplary process executions, which are afterwards analyzed in detail with regard to undesired behaviour such as contradictions, deadlocks or deviations from the behaviour in reality. Often, for a better understanding of a model, also counterexamples are explicitly constructed to verify whether a model prevents a particular behaviour [16]. For generating exemplary process executions it is necessary to execute declarative process models. In [3] both MP-Declare templates and Declare templates are translated into the logic language Alloy[1] 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 [22] also use Alloy to generate event logs. In [16], 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 behaviours. 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 lenth required to find out whether process models are equivalent is still there and this issue has not been solved so far.

4 Determining an upper bound for Model Checking

In this section we determine and prove an upper bound for the trace length to check two Declare models for equality. The main idea is to transform each of the process models to be compared into a deterministic finite state automaton. This automaton is constructed as a minimized product automaton of the corresponding automata representing the constraints of a process model. Hence, we transform the problem of checking two process models for equality into the language equivalence problem of deterministic finite state automata. This transformation allows us to give a mathematical proof that the trace length which must be considered depends on the product of the number of states of the corresponding product automata. In the following we first introduce deterministic finite state automata and explain the transformation of Declare templates into such automata (cf. Section 4.1). Afterwards we show the construction of the

[1] https://alloytools.org/



(a) notRespondedExistence (b) notResponse

Fig. 3: FSA for additional Declare templates

minimal product automaton for a process model (cf. Section 4.2). In Section 4.3 we prove our theorem and determine the theoretical upper bound for the trace length. In Section 4.4 based on this theorem, we formulate a general model checking algorithm and apply it to our running example. In Section 4.6 we discuss consequences of this theorem and the expandability to other declarative process modelling languages.

4.1 Transformation of Declare Templates into Finite State Automata

The first step of our approach is to transform the Declare templates (cf.Section 2) into deterministic finite state automata (FSA) [14]. Therefore, we need some formal definitions:

Definition 4. *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 5. *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*

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

Definition 6. *For a FSA $M = (\Sigma, S, s_0, \delta, F)$ we define the **extended state-transition function** $\hat{\delta} : S \times \Sigma^* \to S$,*

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

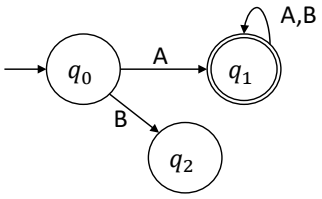


Fig. 4: Finite state automaton M with $\mathcal{L}(M) = \{A\omega \mid \omega \in \{A, B\}^*\}$

In the following, for the sake of simplicity, δ always denotes the extended state-transition function $\hat{\delta}$ for words $\omega \in \Sigma^*$.

Definition 7. *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.*

Example 1. Consider $\Sigma = \{A, B\}$. Then $\Sigma^* = \{\epsilon, A, B, AA, AB, BA, BB, \dots\}$ consists of all strings including any number of A's and B's.
$L := \{A\omega \mid \omega \in \Sigma^*\} = \{A, AA, AB, AAA, \dots\}$ is the language of all words with A at the beginning. The corresponding FSA is depicted in Figure 4.

In [24], the authors present a transformation of most of the Declare templates into a FSA where Σ is the set of the occurring activities. We have determined the corresponding automata of the remaining Declare templates. The results are illustrated in Figure 3.
The traces that fulfill a Declare template are exactly the elements of the language of the corresponding FSA. For example, the trace $t_1 = \langle A, A\rangle$ fulfills the *notResponse* template whereas the trace $t_2 = \langle A, A, B\rangle$ does not. The same thing holds for the automaton, too: t_1 is accepted and t_2 is not accepted (see Figure 3).
The transitions labelled with *:otherwise* are needed because in general there are more than two activities in a process model. The other activities, that do not concern the corresponding template, do not influence the properties of the automata but must be included in the construction of the automata. Details will be described later.

4.2 Transformation of Declare Process Models to Finite State Automata

In this section we show how a Declare process model can be transformed into a finite state automaton. As a Declare process model M consists of a set of different Declare templates $\{T_1, \dots, T_n\}$, a trace t that satisfies M is a trace, that satisfies all the templates:

$$t \text{ satisfies } M \iff t \text{ satisfies } T_1 \wedge \dots \wedge T_n \tag{1}$$

In order to transform a Declare model into a FSA, we apply the concept of the *product automaton* [14]:

Template	$\|S\|$	Template	$\|S\|$
existence(A)	2	alternatePrecedence(A, B)	3
absence(A)	2	chainPrecedence(A, B)	3
atLeast(A, n)	$n+1$	succession(A, B)	3
atMost(A, n)	$n+1$	chainSuccession(A, B)	3
init(A)	3	alternateSuccession(A, B)	3
last(A)	2	notPrecedence(A, B)	3
respondedExistence(A, B)	3	notRespondedExistence(A, B)	4
response(A, B)	2	notResponse(A, B)	3
alternateResponse(A, B)	3	notChainResponse(A, B)	3
chainResponse(A, B)	3	choice(A, B)	2
precedence(A, B)	3	exclusiveChoice(A, B)	4
coExistence(A, B)	4	notChainPrecedence(A, B)	3
notCoExistence(A, B)	4		

Table 2: Number of states of all automata for corresponding Declare templates.

Definition 8. *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 automata 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 automata M_1 and M_2 follows that M accepts exactly the section of $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ [14]:

Remark 1. Let $M_1 = (\Sigma, S_1, s_{0_1}, \delta_1, F_1)$ and $M_2 = (\Sigma, S_2, s_{0_2}, \delta_2, F_2)$ be two deterministic finite-state automata over the same set of symbols Σ. Then $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.

Together with equation (1) follows: A trace t satisfies a Declare model $M = \{T_1, \ldots, T_n\}$ if and only if $t \in \mathcal{L}(M_1) \cap \cdots \cap \mathcal{L}(M_n) = \mathcal{L}(M_1 \times \cdots \times M_n)$ where M_i is the corresponding FSA of T_i.

4.3 Determining an upper bound

In this section we present a method for comparing two Declare process models. The essential part of our approach is the following theorem:

Theorem 1. *Let* M_1 *and* M_2 *be two FSA's with* m *states and* n *states. Then* $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ *if and only if* $\{\omega \in \mathcal{L}(M_1) \mid |\omega| < mn\} = \{\omega \in \mathcal{L}(M_2) \mid |\omega| < mn\}$

Proof. We prove the two directions of the implication. As $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, the equality holds for all subsets. That implies especially that $\{\omega \in \mathcal{L}(M_1) \mid |\omega| < mn\} = \{\omega \in \mathcal{L}(M_2) \mid |\omega| < mn\}$.
We prove the opposite direction by contrapositive. So suppose $\mathcal{L}(M_1) \neq \mathcal{L}(M_2)$ and let a be a word of minimal length with $a \notin \mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \mathcal{L}(M_1 \times M_2)$.
We further define $M := M_1 \times M_2$ as the product automaton of M_1 and M_2.
We assume by contradiction that $|a| \geq mn$. Regard $X := \{\delta(q_0, b) \mid b \text{ prefix of } a\}$.

Since $|X| \geq mn + 1$ and $|S_M| = mn$, there exist two prefixes u and u' of a with $\delta_M(q_0, u) = \delta_M(q_0, u')$. 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 $\delta_M(\delta_M(q_0, u), v) = \delta_M(q_0, u)$ says that v leads M through a loop from state $\delta_M(q_0, u)$ into itself. So we have found a word uz with $\delta_M(q_0, uz) = \delta_M(q_0, a)$ with $|uz| < |a|$. This is a contradiction to the minimality of a.

In order to compare Declare process models, we need to know the number of states S of the corresponding deterministic finite-state automaton of each Declare template. For example, the corresponding automaton of the *notRespondedExistence* template comprises four states ($|S| = 4$). The numbers of states of all Declare templates are shown in Table 2.

We now apply Theorem 1 to our running example. We first calculate the number of states of proces model M_1. The corresponding FSA of the defined template $\mathbf{G}(A \rightarrow \mathbf{X}(B) \wedge \mathbf{X}(\mathbf{X}(C)))$ comprises 4 states. So for M_1 we get a total of $4 \cdot 3 \cdot 3 \cdot 2 = 72$ states. Analogously for M_2 we get $3 \cdot 3 \cdot 3 \cdot 3 = 81$ states. Our theorem says that we need to run all traces t with $|t| < 72 \cdot 81 = 5832$ in order to check the models M_1 and M_2 for equality.

4.4 Checking Declare Models for Equality

Based on the previous results it is now possible to describe an algorithm for checking equality of two Declare process models (cf. Alg. 1). Therefore, we assume that in the two process models to be compared the activities are named identically, otherwise we consider the two models to be different. For both process models, we first transform the constraints of the process models into a minimal FSA and afterwards we construct the product automaton of these single FSAs. Hence, our process model is described by the product automaton. We use the Hopcroft Algorithm [15] for minimization of the product automaton, since this algorithm works very efficiently. To get a smaller upper bound, it is necessary to minimize this automaton. The product of the number of states of both automatons determines the upper bound. Note, that the explicit construction of the product automaton is only necessary if we want a smaller upper bound, otherwise it would be sufficient to multiply the number of states of the FSAs of the constraints to get an upper bound. Afterwards, the determined upper bound is used to configure trace generators for Declare process models, such as proposed in [22,16]. In the typical use cases of trace generators the generated traces possess a trace length that ranges between 0 and a few hundreds. However, from the technical perspective these tools are not limited with regard to the trace length. Hence, they are suited for this task but require more computational power. Eventually, the set of generated traces must be compared for equality. If the two sets are identical, we can say that the process models are equal, too. Otherwise the process models differ.

Algorithm 1: Check the equality of two Declare models

Input: Declare Process Models $P_1 = (A_1, \mathcal{T}_1)$ and $P_2 = (A_2, \mathcal{T}_2)$
Output: True if the models are equal, otherwise False

if $A_1 \neq A_2$ **then**
  return False
else
  $A \leftarrow A_1$
  /* Generate minimal FSA for P_1 and P_2 */
  $U_1 \leftarrow \emptyset$
  for $t \in \mathcal{T}_1$ **do**
    $(A, S_t, s_{0_t}, \delta_t, F_t) \leftarrow$ transform t to minimal FSA
    $U_1 \leftarrow U_1$.add$((A, S_t, s_{0_t}, \delta_t, F_t))$
  end
  $(A, S_{U_1}, s_{0_U 1}, \delta_{U1}, F_{U1}) \leftarrow$ create minimal product automaton of U_1
  $n \leftarrow |S_{U1}|$
  $U_2 \leftarrow \emptyset$
  for $t \in \mathcal{T}_2$ **do**
    $(A, S_t, s_{0t}, \delta_t, F_t) \leftarrow$ transform t to minimal FSA
    $U_2 \leftarrow U_2$.add$((A, S_t, s_{0_t}, \delta_t, F_t))$
  end
  $(A, S_{U_2}, s_{0_U 2}, \delta_{U_2}, F_{U_2}) \leftarrow$ create minimal product automaton of U_2
  $m \leftarrow |S_{U_2}|$
  /* Calculating upper bound -1 due to $<$ in Theorem 1 */
  upperBound $\leftarrow m \cdot n - 1$
  /* Generate traces until upperBound */
  tracesP$_1$ $\leftarrow$ generate all traces for P_1 with length $\leq$ upperBound
  tracesP$_2$ $\leftarrow$ generate all traces for P_2 with length $\leq$ upperBound
  /* Comparing the generated traces */
  if tracesP$_1$ $\neq$ tracesP$_2$ **then**
    return False
  else
    return True
  end
end

4.5 Evaluation and Runtime Analysis

We discuss the execution time of our algorithm by determining its asymptotic behaviour. For the construction of the non-mimized product automaton from the constraints of a process model we get an execution time of

$$\mathcal{O}\left(\prod_{t \in \mathcal{T}_1} |S_t|\right).$$

The product automaton can be minimized in [15]:

$$\mathcal{O}\left(\prod_{t \in \mathcal{T}_1} |S_t| \cdot \log\log\left(\prod_{t \in \mathcal{T}_1} |S_t|\right)\right).$$

The most computational intensive task is the generation and checking of the traces. In dependency of the applied technique (i.e. SAT solving) the execution time differs. We denote this execution time in dependency of the considered process model $\mathcal{P}$ with $\gamma(\mathcal{P})$. SAT solving for propositional logic is known to be NP-complete (Cook–Levin theorem [7]). Hence, the execution time for generating and validating traces is exponential and also dominates the execution time of our algorithm. Note that measuring the execution time does not primarily evaluate the algorithm itself rather than the applied SAT solver. Hence, we have in summary the following asymptotic behaviour for our algorithm, where the first two terms describe the execution time of constructing the corresponding non minimal product automata, the third and fourth term the minimization of the two product automata, and the last two terms describe the execution time for generating and checking the traces until the upper bound:

$$\mathcal{O}\left(\prod_{t\in\mathcal{T}_1}|S_t|\right)+\mathcal{O}\left(\prod_{t\in\mathcal{T}_2}|S_t|\right)+\mathcal{O}\left(\prod_{t\in\mathcal{T}_1}|S_t|\cdot\log\log\left(\prod_{t\in\mathcal{T}_1}|S_t|\right)\right)$$
$$+\mathcal{O}\left(\prod_{t\in\mathcal{T}_2}|S_t|\cdot\log\log\left(\prod_{t\in\mathcal{T}_2}|S_t|\right)\right)+\gamma(\mathcal{P}_1)+\gamma(\mathcal{P}_2).$$

Since the last terms are the predominately ones the execution time of our algorithm is exponential. However, if we are only interested in the upper bound, the execution time is constant, since both the explicit construction of the product automata as well as the trace generation is unnecessary. We evaluate the development of the upper bound in dependency of the number of constraints in the considered process models. Therefore, we assume the worst case, i.e. that each FSA of the corresponding templates has 4 states, since this is the maximal number of states, beside the recursive templates *atLeast* and *atMost*.

$$\left(\prod_{t\in\mathcal{T}_1}|S_t|\right)\cdot\left(\prod_{t\in\mathcal{T}_2}|S_t|\right)-1\leq 4^{|\mathcal{T}_1|}\cdot 4^{|\mathcal{T}_2|}-1$$

We observe that the upper bound grows also exponentially. In Figure 5 we exemplarily depict the increase of the upper bound in dependency of the number of constraints of two process models.

4.6 Limitations and Expandability to other Process Modelling Languages

Since our upper bound depends only on the ability to transform a process model into a FSA, this concept can be applied to any declarative modelling language whose expressiveness allows a mapping to a FSA. For example, in case of the multi-perspective extension of Declare, MP-Declare, that means that first the templates must be transformed into Colored-Petri-Nets. Afterwards, these Petri Nets can be transformed into a FSA. However, as the small running example already reveals, in general the upper bound is relative high. Hence, it needs a lot of computational power to check process models for equality in this way. However, since the statement of the theorem is an equivalence, this theorem can

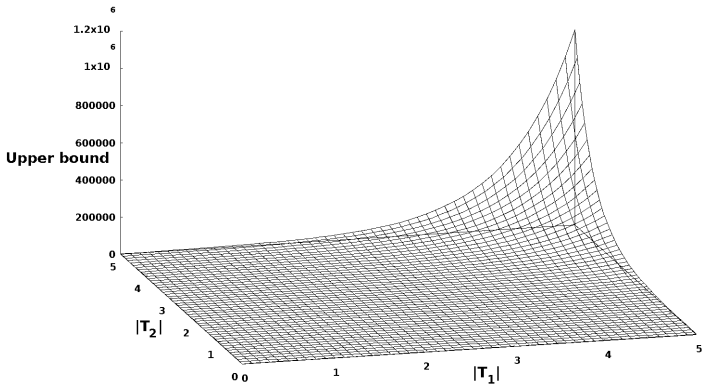


Fig. 5: Development of the upper bound in dependency of the number of constraints of the process models to be compared

be used for searching a counterexample to prove the difference of the models. For this issue, in many cases it will not be necessary to simulate all traces to the upper bound to verify the difference of models.

5 Conclusion and Future Work

In this paper we determined and prove an upper bound for the trace length for comparing two Declare process models for equality by using FSA construction and minimization. We used this upper bound for formulating a model checking algorithm and analyzed its execution time in detail. The algorithm shows an exponential execution time. Also the upper bound increases exponentially in dependency of the number of constraints. In future work, it will be investigated whether a significant smaller upper bound can be found by considering the process specific characteristic of the Declare templates. It should be also investigated whether a probabilistic upper bound, that guarantees equality with a particular probability, can be found. An upper bound for further modelling languages, like MP-Declare, should also be determined.

References

1. Aalst, W.M.P.v.d.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer, Wiesbaden (2011)
2. Abbad, A., Burattin, A., Slaats, T., Petersen, A.C.M., Hildebrandt, T.T., Weber, B.: Exploring the understandability of a hybrid process design artifact based on dcr graphs. pp. 69–84. Springer (2019)
3. Ackermann, L., Schönig, S., Petter, S., Schützenmeier, N., Jablonski, S.: Execution of multi-perspective declarative process models. In: OTM 2018 Conferences
4. Andaloussi, A.A., Buch-Lorentsen, J., Lopez, H.A., Slaats, T., Weber, B.: Exploring the modeling of declarative processes using a hybrid approach. In: Proc. of 38th Int. Conf. on Conceptual Modeling 2019. pp. 162–170. Springer (2019)
5. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Systems with Applications (2016)

6. Ciccio, C.D., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017)
7. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. of the Third Annual ACM Symp. on Theory of Computing. STOC '71, ACM, NY, USA (1971)
8. 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)
9. 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)
10. 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)
11. Gastin, P., Oddoux, D.: Fast ltl to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. Springer (2001)
12. Haisjackl, C., Barba, I., Zugal, S., Soffer, P., Hadar, I., Reichert, M., Pinggera, J., Weber, B.: Understanding declare models: strategies, pitfalls, empirical results. Software & Systems Modeling **15**(2), 325–352 (May 2016)
13. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Log. Algebr. Program. **82**(5-7), 164–185 (2013)
14. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Pearson/Addison Wesley (2007)
15. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. Tech. rep., Stanford, CA, USA (1971)
16. Käppel, M., Schönig, S., Ackermann, L., Jablonski, S.: Language-independent look-ahead for checking multi-perspective declarative process models. SoSym (2021)
17. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: Proc. 17th Annual IEEE Symp. on Logic in Computer Science (2002)
18. Pesic, M.: Constraint-based workflow management systems : shifting control to users. Ph.D. thesis, Industrial Engineering and Innovation Sciences (2008)
19. Schönig, S., Ackermann, L., Jablonski, S.: Towards an implementation of data and resource patterns in constraint-based process models. In: Modelsward (2018)
20. Schützenmeier, N., Käppel, M., Petter, S., Schönig, S., Jablonski, S.: Detection of declarative process constraints in LTL formulas. In: EOMAS - 15th Int. Workshop 2019, Selected Papers. LNBIP, vol. 366, pp. 131–145. Springer (2019)
21. Shi, Y., Xiao, S., Li, J., Guo, J., Pu, G.: Sat-based automata construction for ltl over finite traces. In: 2020 27th Asia-Pacific Softw. Eng. Conf. (APSEC) (2020)
22. Skydanienko, V., Francescomarino, C.D., Maggi, F.: A Tool for Generating Event Logs from Multi-Perspective Declare Models. In: BPM (Demos) (2018)
23. 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)
24. Westergaard, M., Stahl, C., Reijers, H.: UnconstrainedMiner : efficient discovery of generalized declarative process models. BPM reports, BPMcenter. org (2013)
25. 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)
26. Zuck, L.: Past temporal logic. Ph.D. thesis, Weizmann Institute, Israel (1986)