From Declarative Processes to Imperative Models

Johannes Prescher, Claudio Di Ciccio, and Jan Mendling*

Vienna University of Economics and Business, Vienna, Austria
{johannes.prescher,claudio.di.ciccio,jan.mendling}@wu.ac.at

Abstract. Nowadays organizations support their creation of value by explicitly defining the processes to be carried out. Processes are specifically discussed from the angle of simplicity, i.e., how compact and easy to understand they can be represented. In most cases, organizations rely on imperative models which, however, become complex and cluttered when it comes to flexibility and optionality. As an alternative, declarative modeling reveals to be effective under such circumstances. While both approaches are well known for themselves, there is still not a deep understanding of their semantic interoperability. With this work, we examine the latter and show how to obtain an imperative model out of a set of declarative constraints. To this aim, we devise an approach leading from a Declare model to a behaviorally equivalent Petri net. Furthermore, we demonstrate that any declarative control flow can be represented by means of a Petri net for which the property of safety always holds true.

1 Introduction

The definition of valid behavior is at the core of every organization in order to support the creation of value. Such behavior is in most cases modeled using an imperative concept, e.g., by means of notations such as Petri nets [1] or BPMN [18]. They explicitly describe the options to continue at each state. However, while imperative approaches are a strong concept when it comes to well-defined processes, they lack clarity once an observed behavior allows for flexible execution. In this case, models following a declarative approach are able to describe the behavior in a more compact way [4].

Recent research, however, acknowledges that hardly any of the available representations would be superior in all circumstances. For instance, it was pointed out that imperative and declarative models are favoring different types of comprehension tasks [19, 31]. Therefore, approaches have been proposed to represent a mined process partly as an imperative model and partly as a declarative model [35]. A problem in this context is, however, to choose which parts would better be shown in either way. In order to allow for an informed decision, a preliminary question has to be answered: is there a possibility to represent the same behavior regardless of the notation?

In this paper, we start answering this research question by describing how to derive an imperative model from a declarative one. We build upon existing work on transformations from Transition Systems to Petri nets by extending the approach to a tool chain

* The research leading to these results has received funding from EU Seventh Framework Programme (FP7) under grant agreement 318275 (GET Service).

that leads from a Declare model to a behaviorally equivalent Petri net. We implemented and tested our approach using the logs of the BPI Challenge from 2013. Lastly, we show that the imperative version always holds the property of safety.

The paper is structured as follows. Section 2 defines the background of our research, namely preliminaries of different representations including automata, transition systems, Petri nets, and Declare. Section 3 defines our transformation approach. Section 4 demonstrates the feasibility of our approach using a prototypical implementation applied to the BPI Challenge 2013. Section 5 discusses related work before Section 6 concludes the paper.

2 Background

In this section, we discuss Finite State Automata as generic, yet verbose representations of behavior. Then, we revisit the essential concepts of Petri nets. Finally, we introduce Declare as a representation based on behavioral constraints.

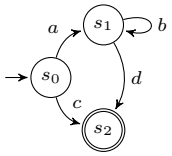


Fig. 1: A process behavior as an FSA

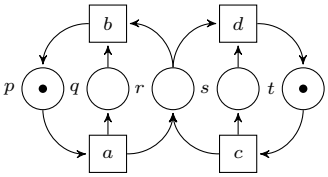


Fig. 2: A Petri net $\mathcal{P}$

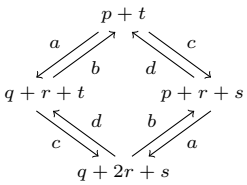


Fig. 3: The Reachability Graph of $\mathcal{P}$

2.1 Finite State Automata

In general, a process can be described as a *stateful* artifact characterized by its conversational behavior, i.e., its potential evolutions resulting from the interaction with some external system, such as a client service. The finite set of possible interactions constitutes the so-called *process alphabet*. The conversational behavior can be represented as a *Finite State Automaton (FSA)*. Its transitions are labeled by process activities, under the assumption that each legal run of the system corresponds to a conversation supported by the process. A process behavior is represented by a finite deterministic Transition System $\mathcal{S} = \langle \mathcal{A}, S, s_0, \delta, S^f \rangle$, where: $\mathcal{A}$ is the process alphabet; S is the finite non-empty set of states; $s_0 \in S$ is the initial state; $\delta : S \times \mathcal{A} \to S$ is the transition function (by $s \xrightarrow{a} s'$ we denote that, from state s, transition a leads to state s'); $S^f \subseteq S$ is the set of final states.

The initial and final states respectively correspond to a legal initialization and termination of the process lifecycle. W.l.o.g., we assume that every state is reachable by traversing the automaton, starting from the initial state. Thus, in Figure 1, the process would admit the instance to either *(i)* perform activity a and then b an arbitrary number of times, and finally d, then terminate, or *(ii)* perform c once and terminate. We can consider FSAs to be for process modeling what Assembly is for computer programming. FSAs are simple and valuable in terms of expressive power, but have problems

modeling concurrency succinctly. Suppose that there are n parallel activities, i.e., all n activities need to be executed but any order is allowed. There are $n!$ possible execution sequences. The FSA thus requires $2n$ states and $n \times 2n - 1$ transitions. This is an example of the well-known "state explosion" problem. Concurrency is known to be well handled by Petri nets.

2.2 Petri nets

Petri nets (PNs) originate from the Ph.D. thesis of Carl Adam Petri [30]. A PN is a directed bipartite graph. Its vertices can be divided into two disjoint finite sets consisting of *places* and *transitions*. Every arc of a PN connects a place to a transition or vice versa, but neither two places nor two transitions can be directly connected. Formally, a Petri net is a tuple $\mathcal{P} = \langle P, T, F \rangle$, where:

- P is a finite set of *place*s;
- T is a finite set of *transition*s;
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*.

Places in a PN may contain a discrete number of marks called *tokens*. Any distribution of tokens over the places represents a configuration of the net called *marking*. Formally, a marking of a PN is a multiset of its places, i.e., a mapping $\mathcal{M} : P \to \mathbb{N}$. We say the marking assigns a number of tokens (graphically represented as black dots) to each place; it represents a state of the system and can be regarded as a vector of non-negative integers of length P. We thus denote a marking M as a linear combination of places, where the linear factor corresponds to the number of tokens in the place. In the following, we will adopt either the vectorial or the polynomial notation (e.g., $M_0 = (1, 0, 0, 0, 1)$ for states p, q, r, s, t in Figure 2, which we also denote as $p + t$) to this extent. A transition $t \in T$ of a PN may *fire* whenever there are sufficient tokens at the start of *all* input arcs; when it fires, it consumes these tokens, and puts tokens at the end of *all* output arcs. Thus, t leads from a marking $M_1 \in \mathcal{M}$ to a marking $M_2 \in \mathcal{M}$ ($M_1 \xrightarrow{t} M_2$). In other words, M_2 is *reachable* from M_1 by means of t. Firings are atomic, i.e., single non-interruptible steps. PNs are always associated to an *initial* marking M_0, denoting the initial status of the described system. The set of all markings reachable from M_0 is called its *reachability set*. A PN with initial marking M_0 is k-bounded iff for every reachable marking M, no place contains more than k tokens (k is the minimal number for which this holds). A 1-bounded net is called *safe*. Figure 2 depicts a 2-bounded PN. A *labeled* Petri net is a PN with labeling function $\lambda : T \to \mathcal{A}$, which puts into correspondence every transition of the net with a symbol (called label) from the alphabet $\mathcal{A}$. Henceforth, we will refer to labeled PNs simply as PNs, for the sake of conciseness.

Thus, modeling a process in terms of a Petri net is rather straightforward: *(i) activities* are modeled by *transition*s; *(ii) conditions* are modeled by *place*s; *(iii) cases* are modeled by *token*s. Figure 2 depicts the parallel evolution of two separate branches of the execution, one involving a loop of c's and d's, the other involving loops of a's and b's.

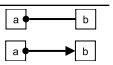
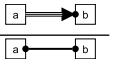
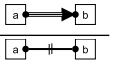
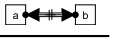
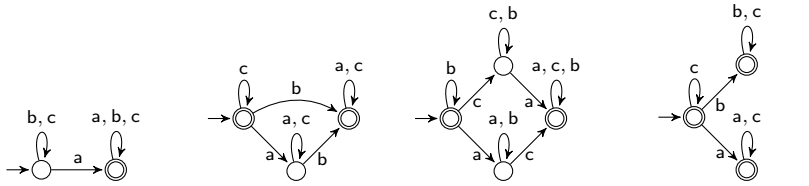
	Constraint	Regular expression	Notation
Existence constraints	$Participation(a)$	`[^a]*(a[^a]*)+[^a]*`	
	$AtMostOne(a)$	`[^a]*(a)?[^a]*`	
	$Init(a)$	`a.*`	
	$End(a)$	`.*a`	
Relation constraints	$RespondedExistence(a, b)$	`[^a]*((a.*b.*)|(b.*a.*))*[^a]*`	
	$Response(a, b)$	`[^a]*(a.*b)*[^a]*`	
	$AlternateResponse(a, b)$	`[^a]*(a[^a]*b[^a]*)*[^a]*`	
	$ChainResponse(a, b)$	`[^a]*(ab[^a]*)*[^a]*`	
	$Precedence(a, b)$	`[^b]*(a.*b)*[^b]*`	
	$AlternatePrecedence(a, b)$	`[^b]*(a[^b]*b[^b]*)*[^b]*`	
	$ChainPrecedence(a, b)$	`[^b]*(ab[^b]*)*[^b]*`	
Mutual relation constraints	$CoExistence(a, b)$	`[^ab]*((a.*b.*)|(b.*a.*))*[^ab]*`	
	$Succession(a, b)$	`[^ab]*(a.*b)*[^ab]*`	
	$AlternateSuccession(a, b)$	`[^ab]*(a[^ab]*b[^ab]*)*[^ab]*`	
	$ChainSuccession(a, b)$	`[^ab]*(ab[^ab]*)*[^ab]*`	
Negative relation constraints	$NotChainSuccession(a, b)$	`[^a]*(aa*[^ab][^a]*)*([^a]*|a)`	
	$NotSuccession(a, b)$	`[^a]*(a[^b]*)*[^ab]*`	
	$NotCoExistence(a, b)$	`[^ab]*((a[^b]*)|(b[^a]*))?`	

Table 1: Semantics of Declare constraints as POSIX Regular Expressions [17]

Reachability Graph and Bisimulation The *Reachability Graph (RG)* of a PN is a Transition System in which *(i)* the set of states is the reachability set (every state is thus a reachable marking), *(ii)* the alphabet coincides with the one of the net, and *(iii)* $M_1 \xrightarrow{t} M_2$ iff there exists a transition t in the net that leads from marking M_1 to M_2. Figure 3 depicts the Reachability Graph for the PN of Figure 2. With a slight abuse of terminology, we will thus refer to the *bisimulation* [27] of a Petri net and a Transition System, meaning that the Reachability Graph of the PN and the Transition System (TS) are bisimilar. We recall here that bisimulation relation is a behavioral equivalence relation, which entails the impossibility for an external user to distinguish the behavior of the two systems. As a consequence, the two systems are trace-equivalent (see [22]).

2.3 Declare Constraints

The need for flexibility in the definition of some types of process has lead to an alternative to the classical "imperative" approach: the "declarative" one. The classical approach is called "imperative" (or also "procedural") because it explicitly represents every step allowed by the process model at hand, by means of transitions (the possible actions to do) among places/states (the legal situations where the process can wait or terminate). This leads to the likely increase of graphical objects as the process allows more alternative executions. The size of the model, though, has undesirable effects on understandability and likelihood of errors – see for instance work on process modeling



(a) $Participation(\mathsf{a})$ (b) $Rspd.Exist.(\mathsf{a},\mathsf{b})$ (c) $CoExistence(\mathsf{a},\mathsf{c})$ (d) $NotCoExist.(\mathsf{a},\mathsf{b})$

Fig. 4: FSAs accepting Declare constraints, on process alphabet $\mathcal{A} = \{\mathsf{a},\mathsf{b},\mathsf{c}\}$

guidelines [25]. In fact, larger models tend to be more difficult to understand [26], not to mention the higher error probability from which they suffer, with respect to small models [24]. Rather than using a procedural language for expressing the allowed sequences of activities, it is based on the description of workflows through the usage of constraints: the idea is that every task can be performed, as long as its execution does not violate any of the specified constraints [29]. Declare [4] is a language defining an extensible set of *templates* for constraints. Declare constraint templates can be divided into two main types: *existence constraints* $\mathcal{C}_E$, and *relation constraints* $\mathcal{C}_R$. The former consists of constraint templates constraining single activities. As such, existence constraints can be expressed as predicates over one variable (the constrained activity): $\mathcal{C}_E(x)$. The latter comprises rules that are imposed on target activities, when activation tasks occur. Relation constraints thus correspond to predicates of arity two: $\mathcal{C}_R(x,y)$. Process alphabet $\mathcal{A}$ is the domain of interpretation for constraints. Given a (possibly empty) set of existence constraints of size $m \geqslant 0$ (resp. relation constraints, of size $n \geqslant 0$) interpreted over alphabet $\mathcal{A}$, each denoted as $\mathcal{C}^{\mathcal{A}}_{E_i}(x)$ (resp. $\mathcal{C}^{\mathcal{A}}_{R_j}(x,y)$), the declarative model consists of their conjunction: $\mathcal{C}^{\mathcal{A}}_{E_1}(x) \wedge \ldots \wedge \mathcal{C}^{\mathcal{A}}_{E_m}(x) \wedge \mathcal{C}^{\mathcal{A}}_{R_1}(x,y) \wedge \ldots \wedge \mathcal{C}^{\mathcal{A}}_{R_n}(x,y)$.

$Participation(\mathsf{a})$ is an existence constraint, which requires the execution of a at least once in every process instance. $AtMostOne(\mathsf{a})$ is its dual, as it specifies that a is not executed more than once in a process instance. $End(\mathsf{a})$ requires that a occurs in every case as the last activity carried out. $RespondedExistence(\mathsf{a},\mathsf{b})$ is a relation constraint. It imposes that if a is performed at least once during the enactment of the process, b must be executed at least once as well, either in the future or in the past, with respect to the time in which a is carried out. $Response(\mathsf{a},\mathsf{b})$ enforces $RespondedExistence(\mathsf{a},\mathsf{b})$ by specifying that b must occur eventually *afterwards*. $AlternateResponse(\mathsf{a},\mathsf{b})$ adds to $Response(\mathsf{a},\mathsf{b})$ the condition that no other a's occur between an execution of a and a subsequent b. Two specializations of the relation constraints are *mutual relation constraints* and *negative relation constraints*. Mutual relation constraints are such that both constrained activities are activation and target. For instance, $CoExistence(\mathsf{a},\mathsf{c})$ is a mutual relation constraint requiring that if a is executed, then c must be performed as well, and vice versa, in any order. Negative relation constraints are such that both constrained activities are activation and target as well. However, the occurrence of one activity *excludes* the occurrence of the other. For instance, $NotCoExistence(\mathsf{a},\mathsf{b})$ is a negative relation constraint imposing that if a is executed, then b cannot be performed at all in the trace, and vice versa.

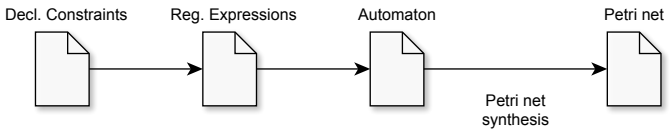


Fig. 5: Obtaining imperative processes as Petri nets from declarative constraints.

$NotSuccession(\mathsf{a}, \mathsf{b})$ is a looser constraint, because it requires that no b's occur after a (and therefore no a's before b). $NotChainSuccession(\mathsf{a}, \mathsf{b})$ requires that the next activity after a cannot be b. An example of graphical representation for a simple Declare process model is drawn in Figure 6a.

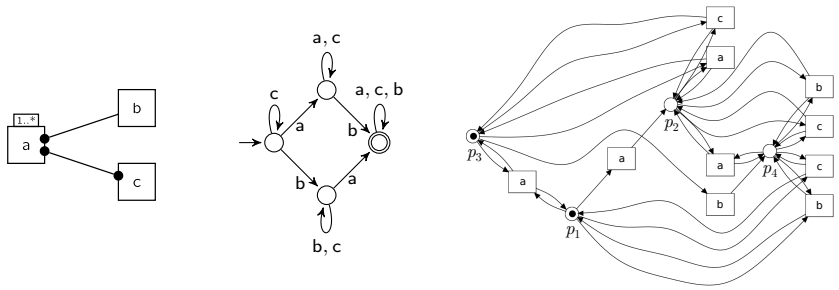
The semantics of Declare templates have been expressed as formulations of several formal languages: as Linear Temporal Logic over Finite Traces (LTL_f) formulas [14], in [13]; as $\mathcal{S}$CIFF integrity constraints [6], in [8]; as First Order Logic (FOL) formulas, interpreted on finite traces, in [16], based on [14]; as Regular Expressions (REs) in [17]. In particular, our work will build upon the last translation, as explained in the next section. Table 1 reports the semantics of Declare constraints as REs. In the table, as well as in the remainder of this paper, we adopt POSIX standard shortcuts for REs, for the sake of brevity. Therefore, in addition to the known Kleene star ($\star$), alternation ($|$) and concatenation ( ) operators, we make use here of *(i)* the `.` and `[^`x`]` shortcuts for respectively matching any character in the alphabet, or any character but x, *(ii)* the + and ? operators for respectively matching from one to any, or none to one, occurrences of the preceding expression. We will also utilize the intersection operator `&` for REs.

3 Conceptual Framework

In this section, we show an approach that describes how to compute a Petri net corresponding to a Declare process model. This approach serves as a conceptual framework for proving that there always exists a Petri net which is *bisimilar* to a Declare process model. Furthermore, the returned PN is proven to be *safe*. The computation consists of three main steps, as sketched in Figure 5.

Declarative constraints to Regular Expressions. We represent all declarative constraints as REs. Each constraint maps to a single RE, i.e., the mapping is one-to-one (cf. Table 1). REs apply to characters. Owing to this, our approach identifies each activity in the process alphabet with a character.

Regular Expressions to Finite State Automaton. The allowed behavior is given by the conjunction of all Declare constraints. Hence, it maps to the *intersection* of the languages accepted by corresponding REs, i.e., the language accepted by the conjunction of the REs (which is in turn a RE itself, being Regular Expressions close w.r.t. the conjunction operation [21]). For the sake of conciseness, though, single REs are thought to directly refer to those activities (characters). They are constrained by the corresponding



(a) Declare (b) FSA (c) PN

Fig. 6: A Declare model, consisting of constraints $\mathit{Participation}(\mathsf{a})$, $\mathit{RespondedExistence}(\mathsf{a},\mathsf{b})$ and $\mathit{CoExistence}(\mathsf{a},\mathsf{c})$, represented with the Declare graphical notation, as a Finite State Automaton and as a Petri net

constraint, disregarding the rest of the process alphabet in their formulation. Consider, for example, $\mathit{Participation}(\mathsf{a})$, depicted in Table 1. The corresponding RE requires the occurrence of a at least once, but also allows any other input beforehand and afterwards. Therefore, we need to limit the set of allowed characters to those which identify activities in the process alphabet $\mathcal{A}$ (see Section 2). This is obtained by means of another RE, which is put in conjunction with the constraint-related ones. This way, we can define the declarative process model described by N constraints by means of a Regular Expression, derived from the conjunction of $N + 1$ REs.

As an example, we consider a process consisting of the following three constraints and having process alphabet $\mathcal{A} = \{\mathsf{a},\mathsf{b},\mathsf{c}\}$:

- $\mathit{Participation}(\mathsf{a})$, translating to `[^a]*(a[^a]*)+[^a]*`, referred to as `(re1)`,
- $\mathit{RespondedExistence}(\mathsf{a},\mathsf{b})$, translating to `[^a]*((a.*b.*)|(b.*a.*))*[^a]*`, referred to as `(re2)`, and
- $\mathit{CoExistence}(\mathsf{a},\mathsf{c})$, translating to `[^a^c]*((a.*c.*)|(c.*a.*))*[^a^c]*`, referred to as `(re3)`.

The mere conjunction of `(re1)`, `(re2)` and `(re3)` would still allow not only for the characters representing activities but also for any input character. In order to limit input characters to those which identify activities in the process alphabet, we thus conjunct the aforementioned three to the following: `([abc]*)`. As a result, the final RE is: `(re1) & (re2) & (re3) & ([abc]*)`.

Continuing with our computation, we transform the RE into the corresponding FSA. We recall here indeed that regular grammars are recognizable through Regular Expressions [9]. Figures 4a to 4c depict the FSAs accepting the languages of `(re1)`, `(re2)` and `(re3)`, respectively. Figure 6b shows the FSA which results from the example we provided. Aside of the transitions that do not change its state, the FSA allows two different runs before reaching its final state, i.e. either $\langle \mathsf{a},\mathsf{b}\rangle$ or $\langle \mathsf{b},\mathsf{a}\rangle$.

Finite State Automaton to Petri net. In the last step of our approach, we derive a Petri net from the FSA. For this purpose, we rely on the theory of regions described in [12], adopted to synthesize PNs from state-based models, such as Transition Systems (and thus, *a fortiori*, FSAs). The rationale behind the theory of regions is to conglomerate sets of places that share the same input and output transitions in common *regions*. The regions translate to places in the derived PN. Input transitions lead to them, and output transitions start from them. In particular, we adopt the approach described in [11], which is proven to return a *safe* Petri net from a Transition System, ensuring the bisimilarity between the two systems (see Section 2.2).

Figure 6c shows the Petri net stemming from the application of the technique of Cortadella et al. [11] to the FSA of our example. Just as the FSA, it contains four places (p_1, p_2, p_3, p_4) and has the initial marking $M = (1, 0, 1, 0)$, i.e., it contains a token in p_1 and p_3. However, the places of the Petri net do not correspond directly to the states of the FSA. Instead (again, without considering the firings that do not change the marking), just as the FSA, the PN allows two different runs ($\langle \mathsf{a}, \mathsf{b} \rangle$ and $\langle \mathsf{b}, \mathsf{a} \rangle$). As the final state of the FSA allows for the execution of any activity in the process alphabet (any character of the input alphabet), the PN also allows for this behavior when its marking is $M = (0, 1, 0, 1)$. Please note that such marking is reachable only by means of the sequence of firings that replicate the sequence of characters leading to the accepting state of the FSA.

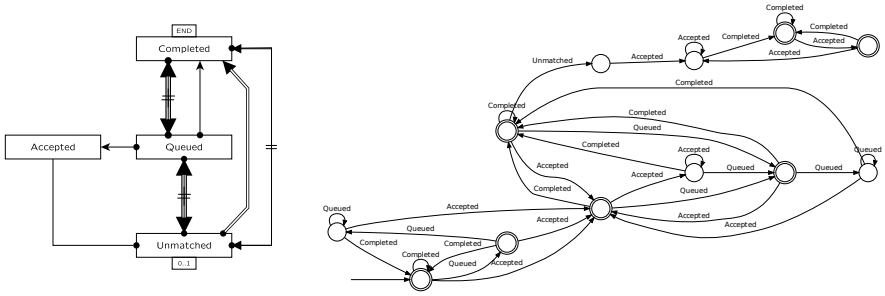
The reader can notice that the returned net presents multiple transitions labeled the same, i.e., representing the same activity. This is due to the fact that label-splitting can be avoided for derived safe PNs only if the Transition System has the property of excitation closure for its transitions, i.e., only if the intersection of those states from which the transitions start can be grouped in one single activating region [11]. However, such property is not guaranteed from the FSAs that Declare processes translate to. Later work of Carmona et al. [7] shows how to balance the trade-off between k-boundedness of the returned Petri net and the number of splitted labels.

To sum up, applying the steps mentioned above, we derive an imperative model from declarative constraints. Note that the operations we perform are transformations that do not alter the behavior. Thus, not only the declarative constraints but also the Regular Expression, the FSA and the PN represent the same behavioral characteristics of the process. Furthermore, we have demonstrated by construction the following theorem.

Theorem 1. *Given any Declare process model $\mathcal{P}_D$ consisting of $n \geqslant 0$ existence constraints and $m \geqslant 0$ relation constraints, expressed over process alphabet $\mathcal{A}$, $\mathcal{P}_D = \bigwedge_{i=1}^{m} \mathcal{C}_{E_i}^{\mathcal{A}}(x) \wedge \bigwedge_{j=1}^{n} \mathcal{C}_{R_j}^{\mathcal{A}}(x, y)$, there always exist a safe Petri net model $\mathcal{P}_N = \langle P, T, F \rangle$ labeled by $\lambda : T \to \mathcal{A}$, which is bisimilar to $\mathcal{P}_D$ and is safe.*

4 Evaluation by Implementation

In this section, we present a feasibility evaluation of our proposed concepts based on a prototypical implementation. We first describe the implementation. Then, we present the results of its application on a Declare model generated from a log of the BPI Challenge 2013. Finally, we discuss insights from the case.



(a) Declare model

(b) Finite State Automaton derived from the Declare model

Fig. 7: The process mined out of BPIC 2013 log [33], as Declare model and FSA

4.1 Implementation

In order to have the opportunity to analyze real-life declarative process models, we have integrated our approach with a tool for the mining of declarative control flows from event logs (see Figure 5), namely MINERful [15]. The MINERful framework comes with an integrated support of a library called *dk.bricks.automaton* [28], for the generation of FSAs out of Regular Expressions. We extended the integrated MINERful-*dk.bricks.automaton* tool in order to make it capable of serializing FSAs into TSML-encoded files. TSML (Transition System Markup Language) is indeed a format supported by ProM, the Process Mining Toolkit [3]. In this way, we have been able to apply the ProM plug-in by van Dongen (see [5]), capable of converting a TSML-encoded Transition System into a Petri net, by using Petrify (see [10]).

4.2 Application to the BPI Challenge 2013

As a real-world data set for validating the approach, we selected the "BPI Challenge 2013, closed problems" log [33] as an application case. For the control-flow discovery task, we have considered the activities' names as their identifiers (Accepted, Completed, Queued and Unmatched). We have set MINERful up in order to return those constraints proven to be valid in every trace (support threshold equal to 1). The discovered model consisted of the following 10 constraints:

$Response$(Queued, Accepted)
$NotChainSuccession$(Queued, Completed)
$Response$(Queued, Completed)
$NotChainSuccession$(Queued, Unmatched)
$Response$(Accepted, Completed)
End(Completed)
$NotSuccession$(Completed, Unmatched)
$AtMostOne$(Unmatched)
$RespondedExistence$(Unmatched, Accepted)
$AlternateResponse$(Unmatched, Completed)

The graphical representation of the model is depicted in Figure 7a. Figure 7b draws the Finite State Automaton derived from the Declare model, and Figure 8 shows the final outcome, as a Petri net. What we can observe from the comparison of the Declare model and the behavior-equivalent Petri net is the multiplication of various activities.

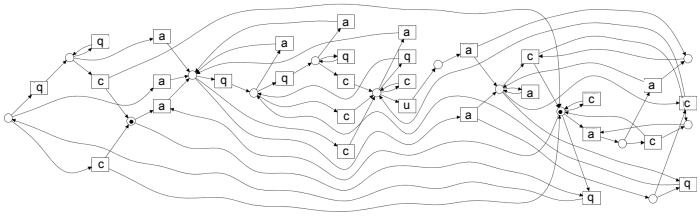


Fig. 8: The Petri net derived from the Finite State Automaton representing the Declare model mined out of BPIC 2013 log [33]. Labels are abbreviated: a = “Accepted”, c = “Completed”, u = “Unmatched”, q = “Queued”.

Although the Declare model seems to be more compact in terms of its nodes and edges, it must be noted that the Petri net is presented as it was produced, i.e., it has not been subject to any post-processing for reducing its complexity. However, the defined chain of transformations provide us with the basis to study the trade-off between compactness of the model and richness of the language in future experiments, conducted on behavior-equivalent Petri nets and Declare models.

5 Related work

The stream of research on the comparison of declarative and imperative modeling approaches has been discussed considering different perspectives. In [32], Pichler et al. investigate both imperative and declarative languages with respect to process model understanding. The issue of maintainability for both languages is discussed in [20]. An open problem for this experimental stream of research has been the question of what a fair comparison is for declarative and imperative models. In this regard, the work of [35] and [36] elaborates on mixed representations as a combination of both approaches from a modeling perspective.

Furthermore, research on automatic process discovery techniques has been defined based on different representations and different techniques of discovery beyond the classical alpha miner [2]. Our selection is not meant to be exhaustive, but rather highlights those approaches that use constraints, automata or transition systems. Van der Aalst et al. propose a two-step approach in [5] in order to discover transition systems which are then synthesized to Petri nets using the “theory of regions”. As well as Van der Aalst et al., Maruster et al. suggested an approach for process discovery in which they deal with noise and imbalance in process logs ([23]). A tool manipulating concurrent specifications, synthesis and optimization of asynchronous controllers is presented in [10]. In order to come up with a better understanding of the mutual strengths and weaknesses of these approaches, De Weerdt et al. ([34]) provide an extensive, multi-dimensional survey of existing process discovery algorithms using real-life event logs. Different representations are, however, not discussed in this survey. In this way, our work provides a basis for an extensive comparison in the future.

6 Conclusion

In this paper, we described an approach to derive imperative process models from declarative process control-flows. To this extent, we utilize a sequence of steps, leading from declarative constraints to Regular Expressions, then to a Finite State Automaton, and finally to a Petri net. We implemented our integrative approach as part of the MINERful software package and evaluated it using the real world case of the BPI Challenge 2013. A remaining limitation is that we do not provide a sound solution for a transformation from an arbitrary imperative model into a declarative representation. In future research, we will address this issue. Furthermore, we plan to utilize the transformation in the design of experiments to study the mutual benefits of PNs and Declare models in model comprehension tasks.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
3. van der Aalst, W.M.P., van Dongen, B.F., Günther, C.W., Rozinat, A., Verbeek, E., Weijters, T.: ProM: The process mining toolkit. In: BPM (Demos) (2009)
4. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM. pp. 1–23 (2006)
5. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software and System Modeling 9(1), 87–111 (2010)
6. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Trans. Comput. Log. 9(4), 29:1–29:43 (2008)
7. Carmona, J., Cortadella, J., Kishinevsky, M.: New region-based algorithms for deriving bounded petri nets. IEEE Trans. Computers 59(3), 371–384 (2010)
8. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. T. Petri Nets and Other Models of Concurrency 2, 278–295 (2009)
9. Chomsky, N., Miller, G.A.: Finite state languages. Information and Control 1(2), 91–112 (1958)
10. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions 80(3), 315–325 (1997)
11. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving petri nets from finite transition systems. IEEE Trans. Comput. 47(8), 859–882 (1998)
12. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Synthesizing petri nets from state-based models. In: Proc. of ICCAD'95. pp. 164–171 (November 1995)
13. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on ltl on finite traces: Insensitivity to infiniteness. In: AAAI (2014)
14. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI (2013)
15. Di Ciccio, C., Mecella, M.: A two-step fast algorithm for the automated discovery of declarative workflows. In: Proc. of CIDM, Singapore 2013. pp. 135–142. IEEE (2013)

16. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Transactions on Management Information Systems (2014)
17. Di Ciccio, C., Mecella, M., Scannapieco, M., Zardetto, D., Catarci, T.: MailOfMine – analyzing mail messages for mining artful collaborative processes. In: Data-Driven Process Discovery and Analysis, vol. 116, pp. 55–81. Springer (2012)
18. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
19. Fahland, D., Lübke, D., Mendling, J., Reijers, H.A., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: BMMDS/EMMSAD. pp. 353–366 (2009)
20. 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 (2009)
21. Gisburg, S., Rose, G.F.: Preservation of languages by transducers. Information and Control 9(2), 153 – 176 (1966)
22. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM 43(3), 555–600 (1996)
23. Maruster, L., Weijters, A.J.M.M., van der Aalst, W.M.P., van den Bosch, A.: A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. Data Min. Knowl. Discov. 13(1), 67–87 (2006)
24. Mendling, J., Neumann, G., van der Aalst, W.M.P.: Understanding the occurrence of errors in process models based on metrics. In: OTM Conferences (1). pp. 113–130 (2007)
25. Mendling, J., Reijers, H.A., van der Aalst, W.M.P.: Seven process modeling guidelines (7PMG). Information & Software Technology 52(2), 127–136 (2010)
26. Mendling, J., Reijers, H.A., Cardoso, J.: What makes process models understandable? In: BPM. pp. 48–63 (2007)
27. Milner, R.: An algebraic definition of simulation between programs. In: IJCAI. pp. 481–489 (1971)
28. Møller, A.: dk.bricks.automaton (2011)
29. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: OTM Conferences (1). pp. 77–94 (2007)
30. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn (1962)
31. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: Business Process Management Workshops (1). pp. 383–394 (2011)
32. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: Business Process Management Workshops (1). pp. 383–394 (2011)
33. Steeman, W.: Real-life event logs – an incident management process: closed problems. Third International Business Process Intelligence Challenge (BPIC'13) (2013)
34. Weerdt, J.D., Backer, M.D., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf. Syst. 37(7), 654–676 (2012)
35. Westergaard, M., Slaats, T.: Cpn tools 4: A process modeling tool combining declarative and imperative paradigms. In: BPM (Demos) (2013)
36. Westergaard, M., Slaats, T.: Mixing paradigms for more comprehensible models. In: BPM. pp. 283–290 (2013)