Rule Propagation:
Adapting Procedural Process Models to Declarative Business Rules

Riccardo De Masellis,
Chiara Di Francescomarino,
Chiara Ghidini
FBK-IRST, Via Sommarive 18,
38050 Trento, Italy

Arne Lapõnin
and Fabrizio Maria Maggi
University of Tartu
Ülikooli 18, 50090 Tartu, Estonia

***Abstract*—**

The debate on advantages and disadvantages of declarative versus procedural process modeling languages for different usage scenarios has been intense. Procedural languages are more suited for describing operational processes while declarative ones for expressing regulations/guidelines, and in many situations the need of combining the benefits of the two rises. Instead of forcing modelers to use a hybrid language, we envisage to keep the two specifications separate and propose a technique that automatically adapts procedural models so as to comply with sets of declarative rules. This not only fits scenarios where, e.g., company processes have to be modified according to changing external rules, but, more in general, it presents a way to take advantage of the flexibility of declarative while maintaining the high level of support provided by procedural languages. Furthermore, by comparing the original and the resulting procedural models, the impact of rules is clearly exposed. In this paper, we frame the problem above by providing its theoretical characterization and propose an automata-based solution, which is then evaluated against approaches leveraging state-of-the-art techniques for process discovery and model repair.

1. Introduction

The dichotomy between procedural and declarative process modeling languages has originated, in the last few years, a stream of comparative investigations (see, e.g., [1], [2]) to better understand their distinctive characteristics and to support the choice of the most suitable paradigm to represent the scenario at hand. While advantages and limitations of the two paradigms are still a matter of investigation, both in academic research and in industry, a trend has emerged to consider hybrid approaches combining a mixture of procedural and declarative specifications. The motivations behind this trend rely on the surmise that many real-life processes are characterized by a mixture of *(i)* less structured processes with a high level of variability, which can usually be described in a compact way using declarative languages such as DECLARE [3] or DCR Graphs [4]; and *(ii)* more stable processes with well structured control flows, which are more appropriate for traditional procedural languages such as Petri nets [5].

Several recent efforts have therefore being devoted both to the automatic discovery of hybrid processes (see, e.g., [6], [7]) and to the proposal of hybrid modeling languages (see, e.g., [8], [9], [10]). Concerning hybrid modeling languages, two different approaches can be observed: a first one devoted to obtain a fully mixed language, where the declarative and procedural notations are almost fused together; and a second one where the declarative and procedural model parts are kept separate so as not to hamper the perceptual discriminability of the various model elements [11]. Examples of the first and second approaches are the BPMN-D language proposed in [8] and the semantics of hybrid languages proposed in [10], respectively.

Our work shares the motivations of [10] to keep the declarative and procedural model parts separate. In fact, we push to the limit the observation that declarative and procedural process modeling languages complement each other, and we target a scenario in which a procedural (say, Petri nets) and a declarative (say, DECLARE) language are used side by side. This point of view is similar to the one of proposals where the BPMN process modeling language and the SRML rule modeling language are used to respectively capture the control flow and the regulatory perspective of a procedure [12]. This gives the modelers the freedom to use the most suitable language for the part of the procedure at hand. In addition, separating procedural and declarative specifications accommodates for situations in which the procedural and declarative parts of the model are actually provided by different parts of an organization, or in which external regulatory constraints need to be applied on top of internal procedural models (think for instance to the adoption of governmental regulations to be applied to process models of an organization).

However, making sense of a model composed of two complete separate parts can be challenging for users. In addition, since nowadays workflow systems are mainly driven by procedural specifications, configuring these systems taking into consideration both a procedural and a declarative model could become very problematic. For this reason, in this paper, we aim at expressing such a combined model

using only the procedural notation. This is obtained by automatically adapting the procedural part so as to comply with the set of declarative rules. In particular, we first frame the problem from a formal standpoint and focus on *consistent* combined models, i.e., models where the declarative and procedural part do not conflict with each other and hence admit a non empty set of intersecting behaviors. Then, we focus on a challenging aspect of this task: more than one change in the procedural model can be made to "solve the problem", and different changes can lead to adapted models showing different characteristics.

We address this challenge in three steps. First, we define a set of heuristics and metrics to guide/evaluate the procedural process adaptation. These heuristics and metrics aim at adapting the procedural models *(i)* keeping the new model syntactically similar to the original one; *(ii)* not adding new behaviors; *(iii)* removing as less behaviors as possible; and *(iv)* preserving block-structured models when applicable. Second, we propose two automata-based techniques to compute the procedural model adaptation: one exploits the automated synthesis of Petri nets from automata, while the other is a novel technique that adapts the original procedural part to accommodate for declarative rules. The latter technique is proposed since the synthesis of Petri nets from automata fails to meet all the metrics listed above. Third, we illustrate a wide experimentation we carried out to compare our automata-based solutions with alternative log-based approaches leveraging state-of-the-art techniques for process discovery and model repair.

In detail, our contributions are the following: *(i)* we propose a combined model that maintains the declarative and procedural components of a process model separate and we describe and define the general problem of adapting procedural models to declarative rules. We use Petri (in fact workflow) nets and DECLARE for the two components of the model due to their well understood formal bases (Section 3); *(ii)* we propose two automata-based techniques for adapting the procedural part (Section 4). The two techniques are implemented in a novel ProM plug-in (Section 4.3); *(iii)* we prove the effectiveness of our solution with a comparison with other approaches based on existing (log-based) techniques (Section 5).

2. Related Work

Recent research has put into evidence synergies between imperative and declarative approaches [1]. Therefore, the combination of conceptual modeling languages to arrive at a hybrid language is pursued based on the idea that such approaches complement each other. Accordingly, hybrid process modeling notations have been proposed. In particular, [8] provides a conservative extension of BPMN for declarative process modeling, namely BPMN-D, and shows that DECLARE models can be transformed into readable BPMN-D models. [9] proposes to extend Coloured Petri nets with the possibility of linking transitions to DECLARE rules directly. The notion of *transition enablement* is extended to handle declarative links between transitions. A recent implementation of this technique is made available in CPN Tools 4.0 [13]. [7] extends the work in [9] by defining a semantics based on mapping Declare rules to R/I-nets and by proposing modeling guidelines for the mixed paradigm. In [10], the authors present a formal semantics for a hybrid process modeling notation. In their proposal, a hybrid process model is hierarchical, where each of its sub-processes may be specified in either a procedural or declarative fashion.

To the best of our knowledge, the problem of adapting procedural process models to declarative rules provides a completely new challenge for the BPM community. In this paper, we approach the problem from a logic-based perspective and propose two initial solutions to it.

3. Definitions

We use Petri nets (PN) [5] to represent procedural process models, as they provide the formal foundations of several procedural languages and are one of the standard ways to model and analyze processes. A PN is a directed bipartite graph with two node types: *places* (graphically represented by circles) and *transitions* (graphically represented by squares) connected via directed arcs.

Definition 1 (Petri net)**.** *A* Petri net *is a triple* (P, T, F) *where* P *and* T *are the set of* places *and* transitions *respectively, such that* $P \cap T = \emptyset$ *and* $F \subseteq (P \times T) \cup (T \times P)$ *is the flow relation.*

Transitions represent activities and places are used to model causal flow relations. The *preset* of a transition t is the set of its input places: ${}^\bullet t = \{p \in P \mid (p, t) \in F\}$ and the *postset* of t is the set of its output places: $t^\bullet = \{p \in P \mid (t, p) \in F\}$. Definitions of pre- and postsets of places are analogous. Places in a PN may contain a discrete number of marks called tokens. Any distribution of tokens over the places, formally represented by a total mapping $M : P \mapsto \mathbb{N}$, represents a configuration of the net called a *marking*.

The expressivity of PNs exceeds, in the general case, what is needed to model business processes, which typically have a well-defined starting point and a well-defined ending point. This imposes syntactic restrictions on PNs, which result in the following definition of a workflow net (WF-net) [5].

Definition 2 (Workflow-net)**.** *A Petri net* (P, T, F) *is a workflow net if it has a single source place* $start$*, a single sink place* end*, and every place and every transition is on a path from start to end, i.e., for all* $n \in P \cup T$*,* $(start, n) \in \overline{F}$ *and* $(n, end) \in \overline{F}$*, where* $\overline{F}$ *is the reflexive transitive closure of* F*.*

The same concept of single-entry-single-exit point for the whole net is a property that can be recursively applied to every net sub-component: the resulting desideratum is the block-structuredness. A WF-net is *block-structured* if for every node with multiple outgoing arcs (a split) there is a corresponding node with multiple incoming arcs (a join), and vice versa, such that the fragment of the model between

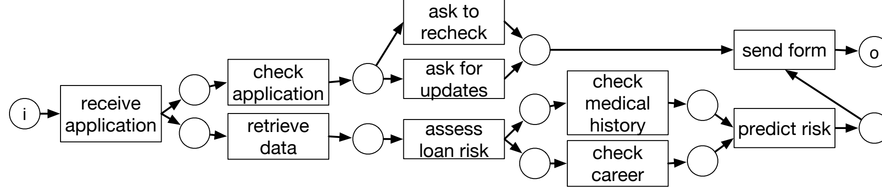


Figure 1: A loan application procedural process.

the split and the join forms a single-entry-single-exit process component [14].

A marking in a WF-net represents the *workflow state* of a single case. The semantics of a PN/WF-net, and in particular the notion of *valid firing*, defines how transitions route tokens through the net so that they correspond to a process execution. A transition $t \in T$ is *enabled* in marking M if each of its input places $^\bullet t$ contains at least one token, i.e., if $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$. An enabled transition t in marking M may *fire* and in the resulting marking M' one token is removed from each of the input places $^\bullet t$ and one token is produced for each of the output places $t^\bullet$. Formally, we say that t is a *valid firing* in M and write $M \xrightarrow{t} M'$ if t is enabled in M and M' is such that, for each $p \in P$: $M'(p) = M(p) - 1$ if $p \in {}^\bullet t \setminus t^\bullet$; $M'(p) = M(p) + 1$ if $p \in t^\bullet \setminus {}^\bullet t$ or $M'(p) = M(p)$ otherwise. We distinguish two special markings: the *initial marking* M_0 such that $M_0(start) = 1$ and $M_0(p) = 0$ for any $p \in P \setminus \{start\}$ and *final marking* M_f such that $M_0(end) = 1$ and $M_0(p) = 0$ for any $p \in P \setminus \{end\}$.

Definition 3 (k-safeness)**.** *A marking of a PN/WF-net is k-safe if the number of tokens in all places is at most k. A PN/WF-net is k-safe if the initial marking is k-safe and the marking of all cases is k-safe.*

From now on, we concentrate on 1-safe WF-nets, which generalize the class of *structured workflows* and are the basis for best practices in process modeling [15]. We also use safeness as a synonym of 1-safeness.

Definition 4 (WF-net language)**.** *Let $W = (P, T, F)$ be a WF-net and T^* the set of all sequences (words) with symbols in T. The* language of W*, i.e., the set of executions accepted by W, is set $\mathcal{L}_W \subseteq T^*$ of net executions $t_1, t_2, \ldots t_n$ for which there exists a sequence of markings $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$ such that: M_0 is the initial marking; for each $i \in \{1 \ldots n\}$, $M_{i-1} \xrightarrow{t_i} M_i$ is a valid firing and $M_n = M_f$ is the final marking.*

As for the declarative language, we focus on DECLARE [3]. Unlike procedural models, where all allowed executions must be explicitly represented, DECLARE models have an open flavor where the agents responsible for the process execution can freely choose how to perform the involved activities, provided that the resulting execution trace complies with a set of rules. Besides, it is grounded on a well-established semantics: given a set of activities T, each DECLARE rule is a Linear-time Temporal Logic (LTL$_f$) formula over T (with $f$inite execution semantics) and the set of allowed finite executions are those satisfying the formulae. Due to page limit, we refer to [16] for the complete syntax and semantics of LTL$_f$. Among all possible LTL$_f$ rules, some specific *patterns* have been singled out as particularly meaningful for expressing DECLARE processes. For instance, let $\mathsf{a}, \mathsf{b} \in T$: $absence(\mathsf{a})$ means that activity a cannot ever be performed; $precedence(\mathsf{a}, \mathsf{b})$ imposes that activity a must precede b; $response(\mathsf{a}, \mathsf{b})$ that each time a is performed, b is eventually performed; $alternateResponse(\mathsf{a}, \mathsf{b})$ means that whenever a is performed, b should eventually be performed with no other a in between and $alternatePrecedence(\mathsf{a}, \mathsf{b})$ imposes that activity a must precede b with no other b in between.

Definition 5 (rule language)**.** *Let T be a set of activities and let Φ be the LTL$_f$ formula obtained as the conjunction of a set of DECLARE rules. The* language of Φ*, i.e., the set of executions compliant with Φ, is set $\mathcal{L}_D \subseteq T^*$ of executions $t_1, t_2 \ldots t_n$ such that $t_1, t_2 \ldots t_n \models \Phi$.*

We now formally define the semantics of our combined models, which are specified by a procedural and a declarative part.

Definition 6 (combined model)**.** *A combined model is a pair $C = (W, \Phi)$ where W is a workflow net and Φ is the LTL$_f$ formula obtained as the conjunction of a set of DECLARE rules. The language $\mathcal{L}_{W \cap D}$ of C is the set of executions accepted by W and compliant with Φ, formally $\mathcal{L}_{W \cap D} = \mathcal{L}_W \cap \mathcal{L}_D$.*

Example 1. *Let the WF-net in Figure 1 describe the happy path of the procedural part of a loan process. In this scenario, the processing of the application is split in two parallel branches: one (the lower) dealing with major check concerning the reliability of the applicant, and the other (the upper) focusing on mainly administrative checks. Let us assume that the bank strategic management office issues some guidelines for their procedures so as to save money and time, which represent the declarative specification of the loan process. In particular, the office decides to (i) eliminate very minor activities such as* ask to recheck *and (ii) ensure that the extremely costly activities* check medical history *and* predict risk *are executed only after* ask for updates *is performed. Such guidelines can be expressed with the*

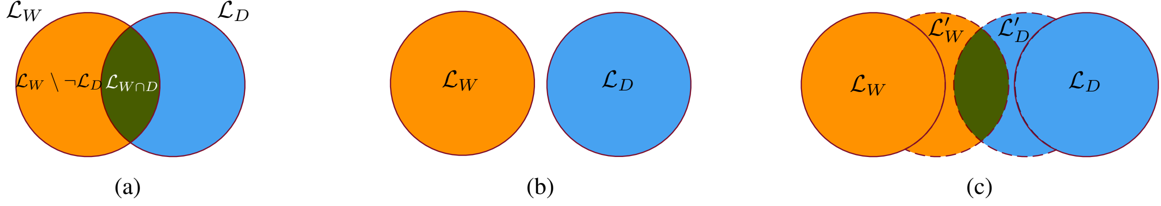


Figure 2: Graphical representation of the relationships between $\mathcal{L}_W$ and $\mathcal{L}_D$.

following sets of DECLARE *rules:*

$$\begin{aligned} &precedence(\textsf{ask for updates}, \textsf{predict risk}),\\ &absence(\textsf{ask to recheck}), \qquad (1)\\ &precedence(\textsf{ask for updates}, \textsf{check medical history}). \end{aligned}$$

The overall model is the combination of the procedural and declarative specifications.

As we already mentioned, our goal is now to propagate the behavior expressed by the declarative part of the combined model to the procedural part. Let $C = (W, \Phi)$ be a combined model, then Figures 2a and 2b graphically show the two situations that may rise: either $\mathcal{L}_W$ and $\mathcal{L}_D$ have common executions, or they do not, respectively. Figure 2a is the case of Example 1. Indeed, the execution

$$\begin{aligned} \pi_1 = (&\textsf{receive application}, \textsf{check application}, \textsf{ask for updates},\\ &\textsf{retrieve data}, \textsf{assess loan risk}, \textsf{check medical history},\\ &\textsf{check career}, \textsf{predict risk}, \textsf{send form}) \end{aligned}$$

satisfies both the WF-net in Figure 1 and the three declarative rules in (1), hence $\pi_1 \in \mathcal{L}_{W\cap D}$. In Figure 2b, all W executions do not conform to rules Φ, hence the model is *inconsistent*. When this happens, the only way to regain consistency is to change/extend one or both specifications in order to get a non-empty intersection, as depicted by Figure 2c. It is important to notice that, by doing this, the semantics of the original net and rules changes deeply, as *new* executions, previously forbidden, are now included. Although such a task of regaining consistency is interesting, we consider it as a separate problem and we focus on the "core" case where the set of net executions compliant with the declarative rules is non-empty.

Given the above, let us assume that the language $\mathcal{L}_{W\cap D}$ of $C = (W, \Phi)$ is not empty. The problem we address is the following: *ideally* find and return a model W' such that the set of accepted executions is $\mathcal{L}_{W'} = \mathcal{L}_{W\cap D}$. In order to guarantee the structural similarity between the original

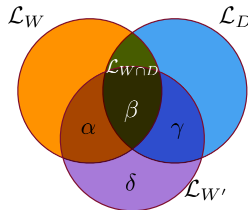


Figure 3: The relationships of languages $\mathcal{L}_W$, $\mathcal{L}_D$ and $\mathcal{L}_{W'}$.

model and the new one (see next section), computing $\mathcal{L}_{W'}$ equal to $\mathcal{L}_{W\cap D}$ may not always be the most appropriate choice, and we may aim at considering $\mathcal{L}_{W'}$ to be "as close as possible" to $\mathcal{L}_{W\cap D}$. This will be done in terms of the relationships between $\mathcal{L}_W$, $\mathcal{L}_D$ and $\mathcal{L}_{W'}$ depicted in Figure 3 in the general case. Indeed, if we relax the assumption $\mathcal{L}_{W'} = \mathcal{L}_{W\cap D}$, the language of the new model $\mathcal{L}_{W'}$ may, in general, contain not only the desired behaviors $\beta = \mathcal{L}_W \cap \mathcal{L}_D \cap \mathcal{L}_{W'}$, but also executions of the procedural not compliant with the declarative $\alpha = \mathcal{L}_W \cap \neg\mathcal{L}_D \cap \mathcal{L}_{W'}$, executions of the declarative not accepted by the procedural $\gamma = \neg\mathcal{L}_W \cap \mathcal{L}_D \cap \mathcal{L}_{W'}$ and even executions not compliant to the declarative nor accepted by the procedural $\delta = \neg\mathcal{L}_W \cap \neg\mathcal{L}_D \cap \mathcal{L}_{W'}$.

4. Automata-based approach

Given a combined model $C = (W, \Phi)$, the problem we want to address in this paper is to find a WF-net W' such that the set of accepted executions $\mathcal{L}_{W'}$ is as close as possible to $\mathcal{L}_{W\cap D}$. We tackle this problem in two separate steps: (1) first, we compute $\mathcal{L}_{W\cap D}$, and then (2) we find the WF-net. We exploit the well-known equivalence between (regular) languages and automata and propose two automata-based approaches to solve both steps. Sometimes, the word *automaton* is used in the literature to refer to a finite-state machine whose accepting condition accommodate for infinite executions. As we focus on finite executions, in the rest of the paper, we use the words automaton and finite-state machine as synonyms.

Definition 7 (Finite-state machine)**.** *A finite-state machine is a tuple $A = (S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is a finite set of symbols, called alphabet, δ is a transition function: $\delta : S \times \Sigma \rightarrow S$, $s_0 \in S$ is the initial state and $F \subseteq S$ is the set of final states.*

Let A be an automaton defined as before, we define the language $\mathcal{L}(A) \subseteq \Sigma^*$ of A as the set of executions $\sigma_1, \sigma_2, \ldots \sigma_n$ for which there exists a path $(s_0, \sigma_1, s_1), (s_1, \sigma_2, s_2) \ldots (s_{n-1}, \sigma_n, s_f)$ in A such that: s_0 is the initial state; for each $i \in \{1, \ldots n\}$, $(s_{i-1}, \sigma_i, s_i) \in \delta$ and $s_f \in F$.

To address step (1), we: *(i)* represent the procedural language $\mathcal{L}_W$ as the reachability graph A_W of W; *(ii)* represent the declarative language $\mathcal{L}_D$ as the automaton A_D for Φ and *(iii)* we exploit the well-known results of automata theory (see, e.g., [17]) to get $\mathcal{L}_{W\cap D}$ as the automaton $A_{W\cap D} = A_W \wedge A_D$ obtained as the automaton synchronous

product $\wedge$ of A_W and A_D. In the following, we will go through each of the above sub-steps in detail.

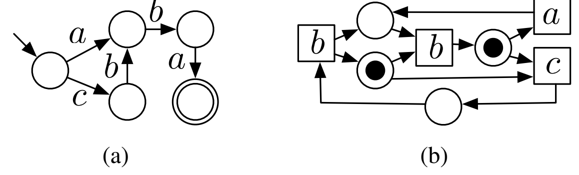
Given a WF-net W, its reachability graph is a graph-like representation of all and only net executions, where nodes/states represent markings and transitions represent firings. In general, such a graph may be infinite-states, as so is the set of reachable markings. However, when considering safe nets, the set of marking is clearly finite, hence their reachability graphs are finite-state machines.

Definition 8 (Reachability graph)**.** *Let $W = (P, T, F)$ be a workflow net. The reachability graph of W is a finite-state machine $A_W = (\mathcal{M}, \Sigma, \delta, M_0, F)$, where: $\mathcal{M}$ is a set of markings; $\Sigma = T$ is the set of activities; $\delta = \mathcal{M} \times T \to \mathcal{M}$ is the transition function; M_0 is the initial marking; $F = M_f$ is the set of final states/markings and $\mathcal{M}$ and δ are defined by mutual induction as the (smallest) set satisfying the following property: if $M \in \mathcal{M}$ then for each valid firing $M \xrightarrow{t} M'$ in W, $(M, t, M') \in \delta$ holds. With notational abuse, we write $M \xrightarrow{t} M'$ for $(M, t, M') \in \delta$.*

It is immediate to see that $\mathcal{L}(A_W) = \mathcal{L}_W$. As for $\mathcal{L}_D$, we build automaton A_D from Φ by exploiting the results and algorithms in [18], which guarantees that $\mathcal{L}(A_D) = \mathcal{L}_D$. Finally, we compute the intersection as the synchronous product [17] of A_W and A_D.

Definition 9 (Automaton intersection)**.** *Let $C = (W, \Phi)$ be a combined model, $A_W = (\mathcal{M}, \Sigma, \delta_W, M_0, F_W)$ be the reachability graph of W and $A_D = (S, \Sigma, \delta_D, s_0, F_D)$ the automaton for Φ. The automaton for the synchronous intersection of A_W and A_D is $A_{W\cap D} = (S_{W\cap D}, \Sigma, \delta_{W\cap D}, (s_0, M_0), F_{W\cap D}, SM)$, where: $S_{W\cap D} = (\mathcal{M} \times S)$ is the set of states; (s_0, M_0) is the initial state; $\delta_{W\cap D} \subseteq S_{W\cap D} \times \Sigma \to S_{W\cap D}$ and it is such that $((M_i, s_i), t_i, (M_i', s_i')) \in \delta_{W\cap D}$ iff $(M_i, t_i, M_i') \in \delta_W \wedge (s_i, t_i, s_i') \in \delta_D$; $F_{W\cap D} \subseteq S_{W\cap D}$ is the set of final states such that $(M_i, s_i) \in F_{W\cap D}$ iff $M_i \in F_W \wedge s_i \in F_D$; $SM : S_{W\cap D} \to \mathcal{M}$ is a projection function selecting the first component, i.e., the marking, from a given state.*

The rest of the section describes step (2), that is, how to generate a WF-net W', whose language will be called $\mathcal{L}_{W'}$, from $A_{W\cap D}$. Our desiderata are: *(i) language similarity* of $\mathcal{L}_{W'}$ to $\mathcal{L}_{W\cap D}$; and *(ii) structural similarity* between W and W' (as a keystone for the understandability of the new procedural model). By structural similarity we do not only mean the graph similarity between W and W' but also the fact that W' preserves some key syntactic properties of W. In this paper, we focus on block-structurdness [14] and no transitions representing the same activity (no duplicate activities) for their well known importance in business process modeling [19]. Unfortunately, the two requirements are usually conflicting: if we want to capture all and only executions in $A_{W\cap D}$, the resulting W' is likely to be chaotic. On the other hand, the language of a simple/understandable model may contain only few executions of $A_{W\cap D}$, and possibly other undesirable ones from sets α, γ and δ as in Figure 3. In order to accommodate different needs, we propose two approaches to solve the problem, one suited to address desideratum *(i)*, language similarity (see Section 4.1), and the other suited to address desideratum *(ii)*, structural similarity (see Section 4.2).



(a) (b)

Figure 4: Simple automaton and its complex synthesized Petri net.

4.1. Petri net synthesis

Although PN synthesis is not novel [20], it is thanks to our formal specification of the problem that it can be successfully employed. Such an approach is grounded on the solid theory of *regions*: a region is a set of automaton states which are somehow similar with respect to incoming and outgoing transitions. In the translation algorithm, each region essentially becomes a place.

Thanks to well-established theoretical foundations, PN synthesis is focused on exactly realizing the language of the input automaton, hence it fulfills the language similarity requirement. However, the resulting net W' in general lacks some other characteristics which are desirable in our scenario, such as: graph similarity with the original net W, block-structuredness and no duplicate activities. As an example, for the simple automaton in Figure 4a, where the leftmost state is the initial one and the double-circled one is the final, the synthesized net[1] in Figure 4b fails in all of the above: it is not a WF-net (it has two start places), it has duplicate activities and it is not block-structured. Figure 5 shows the graphical representation of the WF-net resulting by applying PN synthesis to Example 1. Note that the net is not block-structured.

4.2. ARNE: Automated Rule-to-Net Enactor

For the understandability limitation which PN synthesis suffers from, we implemented a new tool, called ARNE, specifically tailored to return a WF-net W' which: *(i)* is graph similar to the original net W, so as to easily identify the impact of the rules; *(ii)* has no duplicate activities and *(iii)* is block-structured. In order to fulfill the above requirements, we are willing to possibly sacrifice on language similarity. The main idea behind this approach is starting from the original net W and try to remove the behaviors not compliant with the rules instead of building a new net from scratch. Also, we exploit the fact that DECLARE rules essentially express loose precedence relationships between pairs of activities.

1. Obtained using petrify tool, available at: http://www.cs.upc.edu/~jordicf/petrify/

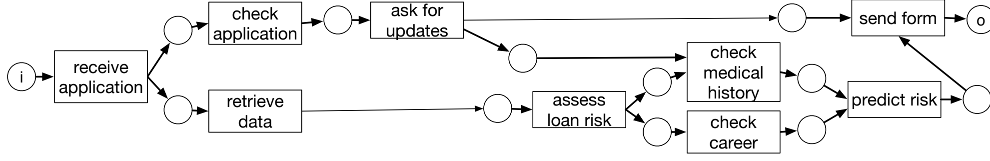


Figure 5: WF-net resulting from the application of PN synthesis to the net and rules in Example 1.

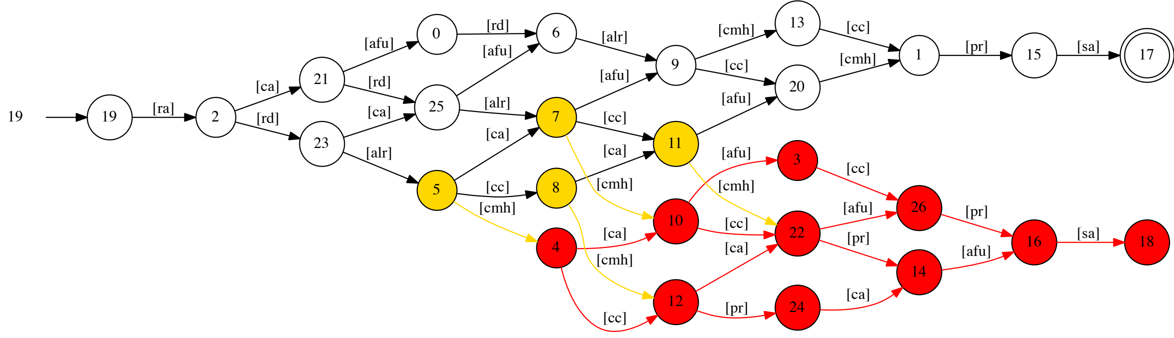


Figure 6: Problem sets of Example 1 after applying steps 1, 2 and 3 of ARNE.

Note that the current implementation of ARNE does not account for some tighter DECLARE rules where the LTL_f temporal operator *next* imposes a direct succession (such as in $chainResponse$), and for cases in which activities involved in loops are mentioned in DECLARE rules, which we plan to support as future work. Although we acknowledge that those are limitations of ARNE, we notice that some of the above have an ambiguous intuitive meaning and require further investigations (for example, the case of a rule specifying that a has to occur immediately after b applied to a net having a inside a loop and b outside).

Let $C = (W, \Phi)$ be a combined model. We now show how ARNE returns a WF-net W' by detailing its steps. We set $W' = W$ and modify W' so as to satisfy the problem specifications.

1 - intersect The reachability graph A_W of W and the automaton A_D for Φ are built. The intersection $A_{W\cap D}$ of the two is computed.

2 - removeUnusedPlaces By comparing the markings of A_W and $A_{W\cap D}$, we identify places in W' that are never reached, and we eliminate them. This essentially amounts to remove branches of exclusive choices that do not comply with Φ.

3 - getProblemSets In this step, we identify the markings and transitions of W' that may violate Φ and cluster them in sets, called *problem sets*, with similar characteristics, so as to take care of each of them separately. Intuitively, a problem set is a set of markings from which, by firing the same transition, the net ends up in a marking violating the rules. We first reason on $A_{W\cap D}$ so as to find *bad* states as explained below, and then, thanks to function SM (associating $A_{W\cap D}$ states to W' markings), we localize the bad markings in W'. Intuitively, every $A_{W\cap D}$ state in $S_{W\cap D}$ from which no path to a final state exists is marked as *bad*. In order to formally define them, we need the reachability relation between states $R \subseteq S_{W\cap D} \times S_{W\cap D}$ as the smallest set satisfying the following properties: $(s, s') \in R$ if $\exists a.(s, a, s') \in \delta_{W\cap D}$ and if $(s, s') \in R \wedge (s', s'') \in R$ then $(s, s'') \in R$. Bad states are then defined as: $B = \{s \in (S_{W\cap D} \setminus F_{W\cap D}) \mid \forall s'.(s' \in S_{W\cap D} \rightarrow (s, s') \notin R)\}$. Also, we define a *semi-bad* state $SB = \{s \in (S_{W\cap D} \setminus B) \mid \exists s', a.(s' \in B \wedge (s, a, s') \in \delta_{W\cap D})\}$. Essentially, every semi-bad state has at least one transition leading to a bad state. Each problem set is the set of semi-bad states sharing at least one common transition to a bad state: $PS_a = \{s \in SB \mid \forall s \in SB, \exists a.(s, a, s') \in \delta_{W\cap D} \wedge s' \in B\}$. Figure 6 shows the problem sets of Example 1 after applying steps 1, 2 and 3 of ARNE. Here, semi-bad states are colored in yellow and bad states in red.

4 - addSyncPoints This step modifies W' by removing behaviors non-compliant with Φ separately for each problem set. Since non-compliant exclusive branches have already been removed by *removeUnusedPlaces*, the ones we tackle here are due to (non-compliant) *interleavings* of two activities in two different branches. We do that by computing *synchronization points* for each problem set: a start synchronization point is a join-transition forcing the execution of two branches to synchronize. The end synchronization point is a set of split-places allowing the parallel execution to continue from where it was left. By analyzing the reachability relation for states in each problem set, we are able

Figure 7: Original net (a) and modified one (b) to accommodate $precedence(b, e)$ with synchronization points.

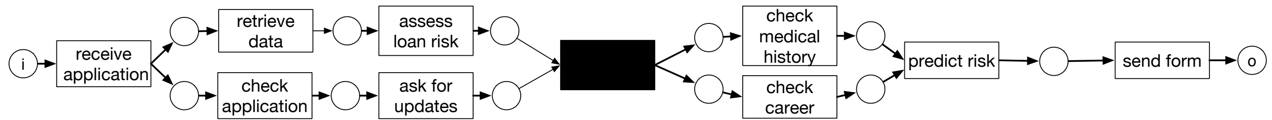


Figure 8: WF-net resulting from the application of ARNE to the net and rules in Example 1.

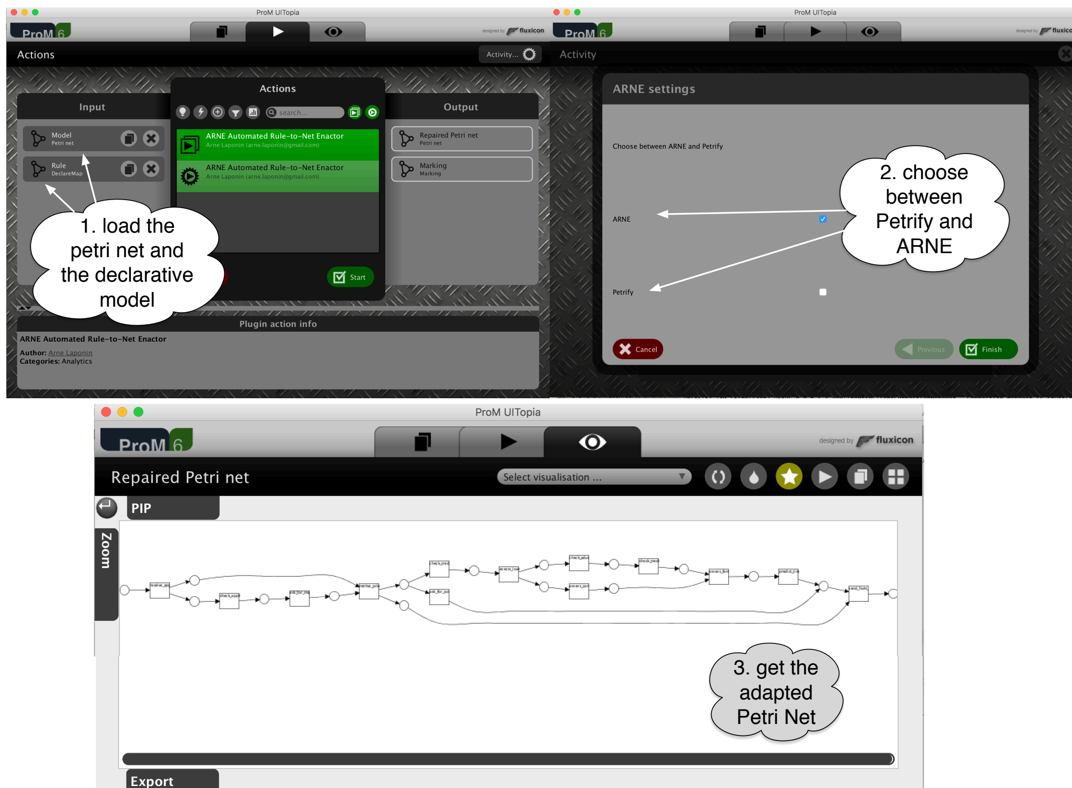


Figure 9: The ARNE *ProM* plug-in.

to identify the location of synchronization points in W'. This allows us to remove as less interleavings as possible but still satisfying the declarative rules. Figure 7 graphically shows how *addSyncPoints* works when dealing with the DECLARE rule $precedence(b, e)$.

5 - flattening When, after adding synchronization points, the model still has non-compliant behaviors, function $flatten$ is used, which essentially "flats" the parallelism by concatenating parallel branches.

We close the section by showing the graphical representation of the WF-net resulting by applying ARNE to Example 1 (Figure 8). We observe that DECLARE rule $absence(\text{ask to recheck})$ results in the elimination of the corresponding exclusive branch (in step $removeUnusedPlaces$). The two $precedence$ rules, instead, are handled by ARNE by synchronizing ask for updates before check medical history and predict risk. Note that the execution

$$\pi_2 = (\text{receive application}, \text{check application}, \text{retrieve data}, \text{assess loan risk}, \text{check career}, \text{ask for updates}, \text{check medical history}, \text{predict risk}, \text{send form})$$

belongs to $\mathcal{L}_{W \cap D}$ but it is not allowed by the PN in Figure 8.

4.3. Implementation

The two automata-based approaches, namely PN synthesis and ARNE, described in Sections 4.1 and 4.2 respec-

	# of Places	# of Transitions	# of Routing		Loops
			place	transitions	
$W1$-XOR	30	40	12	4	No
$W2$-AND	24	26	6	4	No
$W3$-LOOP	16	16	4	2	Yes

TABLE 1: The procedural models

tively, were implemented in a novel *ProM* plug-in.[2] The ARNE algorithm detailed in Section 4.2 was implemented from scratch and it makes use of the FLLOAT library[3] for the automata generation which implements the algorithm in [18]. As for the PN synthesis, we incorporated the existing *petrify* tool.[4] As shown in Figure 9 (details are not meant to be readable) the plug-in takes as inputs a PN and a set of DECLARE rules. Next, the user is prompted with an option of using ARNE or *petrify*, based on whether they want to preserve syntactical similarity or keep all behaviors of the original model. The plug-in returns a repaired process model as a PN. All repair operations presented in Table 2 took less than one second on a standard laptop.

5. Evaluation

We now provide an evaluation of the automata based techniques presented in the previous section. Overall, we did a comparative evaluation between the two automata based procedures (*petrify* and the ARNE algorithm), and two log-based approaches leveraging state-of-the-art techniques for process discovery and model repair. We are interested in answering two research questions concerning the characteristics of the returned model by using the approaches above:

Q1: are the techniques effective in returning WF-nets whose behaviors belong as much as possible to the set of desired behaviors β?

Q2: are the techniques only in β able to satisfy the other problem desiderata, i.e., similarity to the original procedural net, block-structuredness and no duplicate activities?

5.1. Datasets, Procedure and Metrics

For the tool evaluation, we used three different procedural models ($W1$ – $W3$), each Wi paired with different sets of declarative rules. For space limitations, we provide here a brief illustration of the dataset and we leave the interested reader refer to[5] where the combined models (procedural and declarative components) are provided. Table 1 illustrates some characteristics of the procedural models we use. $W3$ distinctive feature is the presence of a loop; $W2$ distinctive feature is a relevant presence of parallelism; and, finally, $W1$ distinctive feature is the relevant presence of alternative branches. These distinctive features are added to the process names (as in Table 1), when relevant to the discussion.

Each procedural model was paired with a number of different sets of DECLARE rules. The aim of the different sets is to challenge the algorithm with a number of distinct DECLARE patterns, combined in different ways and involving transitions placed in different parts of the process. We report below the composition of the three families of rules, one per process. The prefix in the set name makes clear to which process the rules refer to.

$W1$-$D1$ ={*alt. response*, *precedence*}
$W1$-$D2$ ={*alt. precedence*, *response*}
$W1$-$D3$ ={*alt. precedence*, *response*, *absence*}
$W1$-$D4$ ={*precedence*, *response*, *response*}
$W2$-$D1$ ={*response*, *response*}
$W2$-$D2$ ={*response*, *precedence*, *absence*}
$W2$-$D3$ ={*response*, *absence*, *precedence*}
$W2$-$D4$ ={*response*, *alt. response*, *precedence*}
$W3$-$D1$ ={*precedence*}
$W3$-$D2$ ={*response*}
$W3$-$D3$ ={*alt. response*}

Thus, our dataset is composed of 11 entries, one for each suitable combination of the three procedural models in Table 1 with the declarative rules above. For instance, $W1$ has been used in four entries, one for each pair from $W1$-$D1$ to $W1$-$D4$. Similarly for the others.

For each (W, D) of our dataset, we carried out the following steps:

- we computed two procedural models $W_{petrify}$ and W_{ARNE} using the *petrify* and the ARNE algorithms embedded in the implementation of the ARNE plug-in;
- we generated a set $\Pi_W \subseteq \mathcal{L}_W$ of 2 000 traces for W using the PLG2 tool described in [21]; then, we selected from Π_W the subset $\Pi_W \cap \mathcal{L}_D$ of traces satisfying also the LTL$_f$ formulae corresponding to the DECLARE rules in D. Intuitively, $\Pi_W \cap \mathcal{L}_D$ corresponds to a random set of traces compliant with both W and D. We then discovered the procedural model W_{dscvr} from $\Pi_W \cap \mathcal{L}_D$ using the Heuristic miner [22];
- starting from the set Π_W generated in the previous step, we computed the set Π_D by aligning the traces in Π_W against the declarative rules in D. This step was performed using the tool for declarative trace alignment illustrated in [23]. We then used the Repair Model

2. https://github.com/alaponin/AutomatedRuletoNetEnactorProMPlugin
3. https://github.com/RiccardoDeMasellis/FLLOAT
4. http://www.cs.upc.edu/~jordicf/petrify/
5. https://github.com/alaponin/AutomatedRuletoNetEnactorProMPlugin

	W_{dscvr}					W_{rpr}					W_{ARNE}					$W_{\textit{petrify}}$				
Name	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$
$W1$-$D1$	Yes	No	0.55	0.64	T/F/T	Yes	No	0.87	1.00	F/T/T	Yes	No	0.79	1.00	T/T/T	Yes	No	0.80	1.00	T/T/T
$W1$-$D2$	Yes	No	0.57	0.86	T/F/T	Yes	No	0.85	1.00	F/T/T	Yes	No	0.83	1.00	T/T/T	Yes	No	0.79	1.00	T/T/T
$W1$-$D3$	Yes	No	0.58	1.00	T/F/F	Yes	No	0.85	1.00	F/T/T	Yes	No	0.81	1.00	T/T/T	Yes	No	0.74	1.00	T/T/T
$W1$-$D4$	Yes	No	0.57	1.00	T/F/T	Yes	No	0.84	1.00	F/T/T	Yes	No	0.78	1.00	T/T/T	Yes	No	0.71	1.00	T/T/T
$W2$-$D1$	No	No	0.60	0.00	T/T/T	Yes	No	0.82	1.00	F/T/T	Yes	No	0.79	0.15	T/T/T	No	No	0.74	1.00	T/T/T
$W2$-$D2$	No	No	0.59	0.00	T/F/T	Yes	No	0.80	1.00	F/T/T	Yes	No	0.76	0.01	T/T/T	No	Yes	0.74	0.54	T/T/T
$W2$-$D3$	No	No	0.62	0.04	T/F/F	Yes	No	0.83	1.00	F/T/T	Yes	No	0.79	0.01	T/T/T	No	No	0.74	1.00	T/T/T
$W2$-$D4$	Yes	No	0.60	0.06	T/F/F	Yes	No	0.80	0.96	F/T/T	Yes	No	0.79	0.01	T/T/T	No	No	0.73	1.00	T/T/T
$W3$-$D1$	No	No	0.62	0.03	F/F/F	No	No	0.85	1.00	F/T/T	–	–	–	–	–	No	Yes	0.79	1.00	T/T/T
$W3$-$D2$	No	No	0.61	0.12	T/F/F	No	No	0.88	1.00	F/T/T	–	–	–	–	–	No	Yes	0.69	1.00	T/T/T
$W3$-$D3$	No	No	0.62	0.10	F/F/F	No	No	0.82	1.00	F/T/T	–	–	–	–	–	No	Yes	0.75	0.11	T/T/T

TABLE 2: The results.

(remove unused parts) ProM plug-in [24] to repair W with respect to Π_D, thus obtaining a repaired W_{rpr};
- for each returned model $W_{new} \in \{W_{\textit{petrify}}, W_{\text{ARNE}}, W_{dscvr}, W_{rpr}\}$, we performed the following measurements:
  - manually check whether the returned models still satisfy the block-structured and the no-duplicates properties;
  - measure the edit distance between the returned models and the original procedural model;
  - evaluate the percentage of desirable behaviors in β retained by the returned models with respect to the original W;
  - check whether the returned models also admit non-desirable behaviors in α, γ, or δ.

The net similarity between the returned models and the original procedural model was measured using the Graph Edit Distance Similarity ProM plug-in [25], while the measures concerning the behaviors in α, β, γ, and δ where computed using automata. Indeed, given the language-theoretical equivalences between languages and automata, we have that (cfr. Section 3): $\alpha = \mathcal{L}(A_W \wedge \neg A_D \wedge A_{W_{new}})$, $\beta = \mathcal{L}(A_W \wedge A_D \wedge A_{W_{new}})$, $\gamma = \mathcal{L}(\neg A_W \wedge A_D \wedge A_{W_{new}})$ and $\delta = \mathcal{L}(\neg A_W \wedge \neg A_D \wedge A_{W_{new}})$ where $A_{W_{new}}$ is the reachability graph of W_{new}, $\wedge$ is the automata synchronous product operation and $\neg$ is the automata negation operation (simply switching final and non-final states).

- To check whether the returned models also admit non-desirable behaviors in α, γ, or δ, we introduce, with notational abuse, three boolean metrics α, γ, and δ that are set to *true* (T) when the corresponding automaton is empty, i.e., it does not accept any execution, or false (F) otherwise. E.g., if $A_\alpha = A_W \wedge \neg A_D \wedge A_{W_{new}}$ is empty, it means that $\mathcal{L}(A_\alpha) = \alpha = \emptyset$.
- To evaluate the percentage of desirable behaviors in β retained by each returned model W_{new}, we used the following metric (again, with notational abuse):

$$\beta = \frac{|(A_W \wedge A_D) \wedge A_{W_{new}}|}{|A_W \wedge A_D|}$$

where $|A|$ counts the number of different paths of automaton A considering each loop (if present) only once. The above metrics essentially computes the number of behaviors/executions common to W, D, and W_{new} normalized by the number of original behaviors of both W and D.

5.2. Results

The results of our evaluation are reported in Table 2. The missing results for ARNE for model $W3$ are because it does not yet handle loops, which do appear in $W3$. For each of the four methods, the table includes: whether the returned model is still block-structured (**BS**), it contains duplicate activities (**R**); its (edit distance) similarity with the original PN (**S**); the measurement of retained desired behaviors (β) of the returned model and whether the returned model admits non-desirable behaviors in α, γ, or δ ($\alpha/\gamma/\delta$).

Concerning **Q1** we can observe that automata-based techniques are the only ones able to always guarantee $\alpha = \gamma = \delta = true$. While both failing on this measure, discovery- and repair-based approaches provide fairly different results. Discovery is, in fact, the worst technique to be used to solve this problem: this is somehow not surprising as it does not take into account the original PN, but it is exclusively based on a set of (random and non-exhaustive) execution traces in β. From this the lowest similarity **S**. A possible explanation for the fluctuating metric α–γ is that discovery algorithms attempt to generalize the behaviors extracted from traces: thus, even if the discovery is done using traces satisfying both the procedural and the declarative parts of the original model, the generalization can introduce extra behaviors in $\alpha/\gamma/\delta$ as well as lose part of the ones in β.

The repair based technique instead shows more consistent results. It scores very well in **S**, β, and it only fails in α. However, this is a significant limitation of this approach. Indeed, failing in α means that, in some cases, parts of the original net are not modified at all even if they are not compliant with the declarative rules. For example, the repair-based approach is not able to deal with synchronization of parallel executions so that parallel branches are not modified even if they allow behaviors that are not compliant with the rules.

We can then conclude that ARNE and *petrify* are the only techniques in β (**Q1**). The measurement of similarity (**Q2**), both in terms of syntactic edit distance **S** and behaviors (β), is therefore restricted to ARNE and *petrify*. Concerning **S**, we can observe that it is constantly fairly high for both

ARNE ($0.76 - 0.83$) and *petrify* ($0.71 - 0.80$). Except for $W1$-$D1$, for comparable values of β, ARNE shows a higher value of **S**. The ability to retain all the desired behaviors (β), instead, varies greatly for the different dataset entries: for both techniques, we can observe that it is either extremely low (close to 0) or extremely good (equal to 1). The low values of ARNE in adapting $W2$ can be explained by the fact that this algorithm preserves - by construction – the block-structured property of the net. Comparing ARNE and *petrify* on $W2$, we can observe that the price paid by ARNE in terms of behaviors β to maintain the block-structured and the no-duplicates properties corresponds to a gain in a higher similarity **S** with the original net.

To conclude, our evaluation shows that automata-based techniques are the best suited ones to return a procedural process model which satisfies a set of declarative rules. Our first experiments show that ARNE should be chosen when the priority is given to maintain block-structured processes with no duplicate activities. This results in highly similar WF-nets that nonetheless may lose a not negligible amount of behaviors. *Petrify* may be chosen, instead, when maintaining these two structural properties of the net is not crucial.

6. Conclusions

In this paper, we proposed a language to combine procedural and declarative models while keeping the procedural and the declarative parts separate. In addition, we formulated the complex problem of how to adapt the procedural specification to the declarative one. We tackled this problem with two automata-based approaches: one based on the synthesis of Petri nets and one based on a novel algorithm. The two-automata based approaches have been implemented in a *ProM* plug-in and extensively tested against log-based approaches leveraging state-of-the-art techniques for process discovery and model repair. The results emphasize the soundness of the automata-based approaches, but also bring open questions and an opportunity for future investigations. In particular, the paper provides a new challenging research perspective on how to deal with combined procedural and declarative models which should be further expanded. As future work, we plan to extend the functionalities of ARNE so as to cope with loops and DECLARE rules expressing strong relations between activities such as $chainResponse$, to extend our evaluation to real-life process models, and to investigate the issue of human understandability of the adapted procedural models.

References

[1] H. Reijers, T. Slaats, and C. Stahl, "Declarative modeling-An academic dream or the future for BPM?" in *BPM*, 2013, pp. 307–322.

[2] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," in *BPM Workshops*, 2011, pp. 383–394.

[3] M. Pesic, H. Schonenberg, and W. van der Aalst, "DECLARE: Full Support for Loosely-Structured Processes," in *EDOC*, 2007, pp. 287–300.

[4] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti, "Contracts for cross-organizational workflows as timed dynamic condition response graphs," *J. Log. Algebr. Program.*, vol. 82, no. 5-7, pp. 164–185, 2013.

[5] W. M. P. van der Aalst, "The application of petri nets to workflow management," *Journal of Circuits, Systems and Computers*, vol. 08, pp. 21–66, Feb. 1998.

[6] F. Maggi, T. Slaats, and H. Reijers, "The automated discovery of hybrid processes," in *BPM*, 2014, pp. 392–399.

[7] J. De Smedt, J. De Weerdt, J. Vanthienen, and G. Poels, "Mixed-paradigm process modeling with intertwined state spaces," *Business & IS Eng.*, vol. 58, no. 1, pp. 19–29, 2016.

[8] G. De Giacomo, M. Dumas, F. M. Maggi, and M. Montali, "Declarative process modeling in BPMN," in *CAiSE*, 2015, pp. 84–100.

[9] M. Westergaard and T. Slaats, "Mixing paradigms for more comprehensible models," in *BPM*, 2013, pp. 283–290.

[10] T. Slaats, D. M. M. Schunselaar, F. M. Maggi, and H. A. Reijers, "The semantics of hybrid process models," in *CoopIS*, 2016, pp. 531–551.

[11] D. L. Moody, "The physics of notations: toward a scientific basis for constructing visual notations in software engineering," *IEEE TSE*, vol. 35, no. 6, pp. 756–779, 2009.

[12] M. Zur Muehlen and M. Indulska, "Modeling languages for business processes and business rules: A representational analysis," *Inf. Syst.*, vol. 35, no. 4, pp. 379–390, 2010.

[13] M. Westergaard and T. Slaats, "Cpn tools 4: A process modeling tool combining declarative and imperative paradigms." in *BPM (Demos)*, 2013.

[14] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske, "Maximal structuring of acyclic process models," *Comput. J.*, vol. 57, no. 1, pp. 12–35, 2014.

[15] B. Kiepuszewski, A. H. M. ter Hofstede, and C. J. Bussler, "On structured workflow modelling," in *Seminal Contributions to Information Systems Engineering*. Springer, 2013.

[16] G. De Giacomo, R. De Masellis, and M. Montali, "Reasoning on LTL on finite traces: Insensitivity to infiniteness," in *AAAI*, 2014, pp. 1027–1033.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[18] G. De Giacomo, R. De Masellis, M. Grasso, F. M. Maggi, and M. Montali, "Monitoring business metaconstraints based on LTL and LDL for finite traces," in *BPM*, 2014, pp. 1–17.

[19] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters, "Workflow mining: Current status and future directions," in *CoopIS 2003*, 2003, pp. 389–406.

[20] E. Badouel, L. Bernardinello, and P. Darondeau, *Petri Net Synthesis*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2015.

[21] A. Burattin, "PLG2: multiperspective process randomization with online and offline simulations," in *BPM Demos*, 2016, pp. 1–6.

[22] A. Weijters and W. Aalst, "Rediscovering Workflow Models from Event-Based Data using Little Thumb," *Integrated Computer-Aided Engineering*, vol. 10, no. 2, pp. 151–162, 2003.

[23] G. De Giacomo, F. M. Maggi, A. Marrella, and F. Patrizi, "On the disruptive effectiveness of automated planning for ltl$f$-based trace alignment," in *AAAI*, 2017, pp. 3555–3561.

[24] D. Fahland and W. M. P. van der Aalst, "Model repair – aligning process models to reality," *Inf. Syst.*, vol. 47, pp. 220–243, 2015.

[25] R. M. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, and J. Mendling, "Similarity of business process models: Metrics and evaluation," *Inf. Syst.*, vol. 36, no. 2, pp. 498–516, 2011.