UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Arne Lapõnin

Propagating Changes between Declarative and Procedural Process Models

Master's Thesis (30 ECTS)

Supervisors: Fabrizio Maria Maggi
Riccardo De Masellis
Chiara Di Francescomarino

Tartu 2017

Muutuste üle kandmine deklaratiivsetest protseduursetesse protessi mudelitesse

Lühikokkuvõte:

Debatt protseduuriliste ja deklaratiivsete keelte eeliste ja puuduste üle erinevate kasutusjuhtude korral on olnud tuline. Protseduurilised keeled on sobivamad operatiivsete protsesside modelleerimiseks, deklaratiivsed keeli kasutatakse regulatsioonide/juhiste jaoks. Ometi tekib olukordi, kus on mõistlik kombineerida neid keeli, et saavutada parem tulemus. Selle asemel, et sundida modelleerijaid õppima uusi hübriidkeeli, peame me paremaks kahe spetsifikatsiooni eraldi hoidmist ja pakume välja viisi kuidas protseduurilist mudelit automaatselt muuta nii, et see oleks kooskõlas deklaratiivsete reeglitega. Nõudlus sellise lahenduse jaoks tekib, näiteks kui organisatsioon peab muutma protsesse vastavalt muutuvatele välistele reeglitele. Üldiselt, on nii võimalik ära kasutada deklaratiivsete keelte paindlikust ja hoida kõrgetasemelist tuge, mida pakuvad protseduurilised keeled. Lisaks, võrreldes originaalset ja parandatud mudelit, on võimalik selgelt näha reeglite mõju. Käesolevas lõputöös sõnastame me antud probleemi, loome teoreetilise vundamendi ja pakume välja olekumasinatel põhineva lahenduse, mida me võrdleme olemasolevate lahendustega mudelite parandamiseks ja protsesside avastamiseks.

Võtmesõnad: Mudeli parandamine, Mudeli kontrollimine, Deklaratiivsed ja Protseduurilised mudelid.

CERCS: P175, Informaatika, süsteemiteooria

Propagating Changes between Declarative and Procedural Process Models

Abstract:

The debate on advantages and disadvantages of declarative versus procedural process modelling 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 modellers 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 thesis, we frame the problem above by providing its theoretical characterisation and propose an automata-based solution, which is then evaluated against approaches leveraging state-of-the-art techniques for process discovery and model repair.

Keywords: Model repair, Model checking, Declarative and Procedural models.

CERCS: P175, Informatics, systems theory

Contents

1 Introduction

A business process, as defined in [14], is a collection of activities, performed by people or machines, that bring value to the customer. Business Process Management (abbreviated as BPM) is a field that aims to improve, or more specifically discover, analyse, redesign, execute and monitor those processes. To convey information about the business processes diagrams are used to model the processes. Most of the times, these models are created using procedural modelling languages such as BPMN, YAWL or Petri nets. Business Process Model and Notation (BPMN) is an industry standard for modelling business processes. BPMN consists of activity nodes and control nodes representing them [12]. Petri nets consist of circular places, rectangular transitions and arcs connecting them. While BPMN is very intuitive and understandable for a business analyst, Petri nets have a solid mathematical theory behind it, which is useful for analysis.

On the other side of the modelling spectrum, we have declarative languages, which instead of explicitly modelling control flows, restrict the space of available activities. Declarative process modelling offers more flexibility but less control than procedural modelling. Examples of declarative process modelling languages include DECLARE and DCR graphs, both of which are based on templates representing relations between activities.

Suppose that we have banks that have hundreds of processes, which are modelled using procedural process models and we have parliaments or governments issuing laws and acts of legislation, which influence the everyday life of the banks, yet tend to change frequently. Legislation can be modelled using declarative notation. Both types of models are important for the bank but since they are on the opposite end of the modelling spectrum, it is difficult to integrate the knowledge that they represent.

The contrast between procedural and declarative process modelling languages has originated, in the last few years, a stream of comparative investigations (see, e.g., [27, 24]) 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 characterised 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 [23] or DCR Graphs [17]; and *(ii)* more stable processes with well structured control flows, which are more appropriate for traditional procedural languages such as Petri nets [29].

Several recent efforts have therefore being devoted both to the automatic dis-

covery of hybrid processes (see, e.g., [21, 11]) and to the proposal of hybrid modelling languages (see, e.g., [8, 34, 28]). Concerning hybrid modelling languages, two different approaches can be observed: a first one devoted to obtaining 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 [22]. Examples of the first and second approaches are the BPMN-D language proposed in [8] and the semantics of hybrid languages proposed in [28], respectively.

Our work shares the motivations of [28] to keep the declarative and procedural model parts separate. In fact, we push to the limit the observation that declarative and procedural process modelling 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 the proposals where the BPMN process modelling language and the SRML rule modelling language are used to respectively capture the control flow and the regulatory perspective of a procedure [35]. This gives the modellers 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 organisation, 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 organisation).

However, making sense of a model composed of two completely 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, 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 behaviours. 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. 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. 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 describe and define the general problem (Section 4) of adapting procedural models to declarative rules and we propose a combined model (Section 5) that maintains the declarative and procedural components of a process model separate. We use Petri (in fact workflow) nets and DECLARE for the two components of the model due to their well understood formal bases (Section 2); *(ii)* we propose two automata-based techniques for adapting the procedural part (Section 6). The two techniques are implemented in a novel ProM plug-in (Section 7); *(iii)* we prove the effectiveness of our solution with a comparison with other approaches based on existing (log-based) techniques (Section 8). Additionally, we give an overview of the related work in Section 3 and summarise the work in Section 9.

2 Background

This section starts with the introduction of the field of Process Mining and how it fits into BPM. It is important to have a general overview of the field since we are using some of the concepts for adapting the models and the evaluation of our approaches. Then we introduce the basic concepts of Petri nets, LTL_f, DECLARE and automata. The rest of the thesis will be built upon these concepts.

2.1 Process Mining

BPM lifecycle consists of process identification, discovery, analysis, redesign, implementation, monitoring and controlling [14]. Phases covering identification, discovery, monitoring and controlling are supported by the field called Process Mining. In [30] van der Aalst defines Process Mining as a field between data mining and process modelling that discovers, monitors and enhances processes based on the knowledge derived from the event logs produced by the information systems. Process Mining is categorised into three types: discovery, conformance and enhancement.

Process discovery is used to produce a process model based on a given event log without any previous knowledge. One of the first and simplest algorithms for discovery is called α-algorithm. The main idea lies in finding patterns in the event log and constructing a model based on that information. The drawbacks of the algorithm are that it is not able properly handle complex and infrequent behaviour. More advanced discovery algorithms include heuristic mining, genetic mining, region-based mining and inductive mining. Heuristic mining algorithms build upon the α-algorithm by taking into account the frequencies of the events and handling more complex dependencies. Genetic mining algorithms build the model iteratively and rely on randomisation to generate new alternatives. The idea behind region-based approaches is to use either a transition system or a language to construct the model. While inductive mining techniques use *divide-and-conquer* methods and process trees.

Conformance checking is used to compare a process model with an event log and to see whether the model deviates from the real executions. These deviations can be detected, precisely located and used to detect fraud or inefficiencies in the process or be used as a basis for repairing the model. Conformance checking is done using replay, alignments or footprints. By replaying the event log on the model we understand how much of the behaviour the model captures. Alignments allow us to compare the model and the event log and precisely locate where the two deviate from each other. Same can be done using footprints.

Process enhancement is used to extend or improve an existing model based on an event log. One type of enhancement is model repair, in which case the event log

is used to improve the model so that it would better reflect the reality. Another type of enhancement is called extension, which is used to add information, taken from the log to the model.

2.2 Petri net

We use Petri nets (PN) [29] 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 analyse 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) [29].

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 the split and the join forms a single-entry-single-exit process component [26].

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 $\{p \in P \mid M(p) > 0\} \supseteq {}^\bullet t$. When an enabled transition t in marking M *fires*, the resulting marking M' is such that 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 $p \in {}^\bullet t \backslash t^\bullet$;
- $M'(p) = M(p) + 1$ if $p \in t^\bullet \backslash {}^\bullet t$ 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 $p \in P \backslash \{start\}$ and *final marking* M_f such that $M_0(end) = 1$ and $M_0(p) = 0$ for any $p \in P \backslash \{end\}$.

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 [19]. We also use safeness as a synonym of 1-safeness.

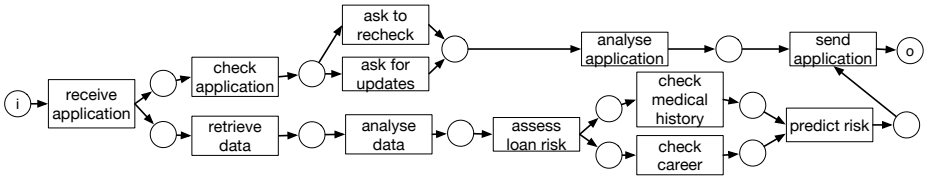

Figure 1: A loan application modelled using Petri net.

Figure 1 shows a 1-safe block-structured WF-net, with a source place represented by i and a sink place by o. When the token is in the state i transition labeled receive application can be fired. Then there are two mutually exclusive branches of which one has to be taken. Upper branch contains an additional XOR gateway, leading to another two mutually exclusive branches, meaning that once the transition ask for updates has been fired, ask to recheck cannot be fired anymore, while the lower one contains an AND gateway, meaning that both transitions check medical history and check career have to be fired.

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.*

2.3 Linear-time Temporal Logic

Before we can introduce declarative modelling languages we have to mention Linear-time Temporal Logic (LTL), introduced in [25], which is a modal temporal logic for encoding the future of paths. LTL is a powerful and flexible method for expressing declarative constraints.

Definition 5 (syntax of LTL)**.** LTL *formulae over the set P of propositional logic are built using the following grammar:*
$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$ *with $a \in P$.*

There are some additional common abbreviations used, including:

- Standard boolean abbreviations, such as *true*, *false*, $\vee$ (or) and $\rightarrow$ (implies).
- $\diamond\varphi$ which is the same as $true\,\mathcal{U}\,\varphi$, intuitively meaning that φ will *eventually* hold.
- $\Box\varphi$ which is the same as $\neg\diamond\neg\varphi$, intuitively meaning that φ will *always* hold.
- $\varphi_1 \mathcal{W} \varphi_2$ which is the same as $(\varphi_1 \mathcal{U} \varphi_2) \wedge \Box\varphi_1$.

LTL was originally developed with an infinite-path semantics, but here we use it for expressing business process executions which eventually terminate, hence we focus on the finite-path variant (LTL_f) defined in [10]. The difference between LTL and LTL_f is in the semantics, the syntax is the same when dealing with either infinite- or finite-paths.

Definition 6 (semantics of LTL_f)**.** *Given a finite trace π, LTL_f formula φ is defined as true at an instant i (for $0 \leq i \leq n$), written $\pi \vDash \varphi$, as:*

- $\pi, i \vDash true$.
- $\pi, i \vDash a$, *for* $a \in P$ *iff* $a \in \pi(i)$.
- $\pi, i \vDash \neg\varphi$ *iff* $\pi, i \nvDash \varphi$

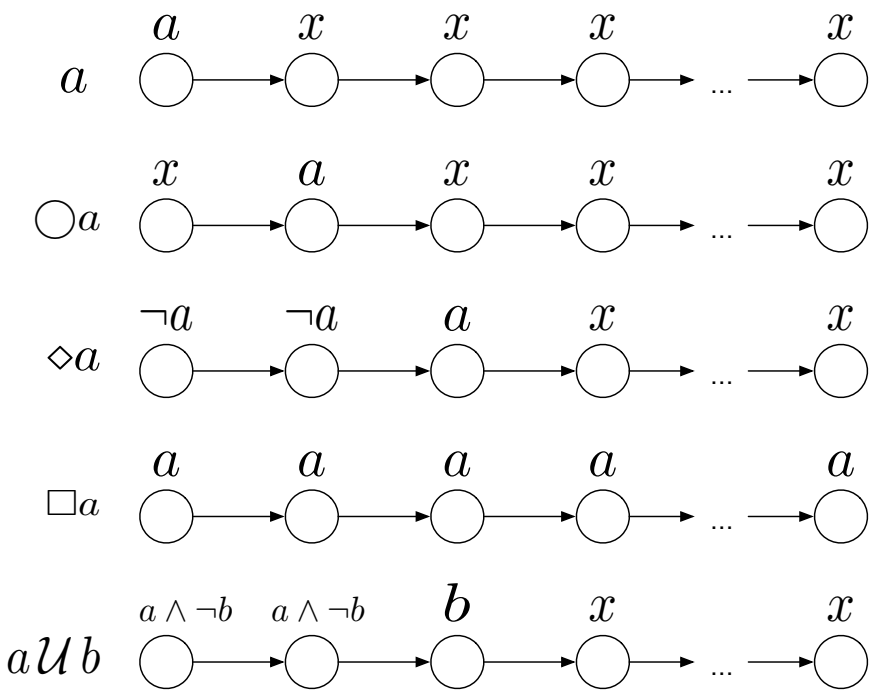

Figure 2: Intuitive semantics of LTL$_f$, with x meaning arbitrary value. Inspired by [3]

- $\pi, i \vDash \varphi_1 \wedge \varphi_2$ *iff* $\pi, i \vDash \varphi_1$ *and* $\pi, i \vDash \varphi_2$.
- $\pi, i \vDash \bigcirc\varphi$ *iff* $i < n$ *and* $\pi, i+1 \vDash \varphi$.
- $\pi, i \vDash \varphi_1 \mathcal{U} \varphi_2$ *iff for some* $j\,(i < j < n)$ *there is* $\pi, j \vDash \varphi_2$ *and for all* $k\,(i < k < j)$ *there is* $\pi, k \vDash \varphi_1$.

We say π satisfies φ, written $\pi \vDash \varphi$, if $\pi, 0 \vDash \varphi$.

Figure 2 presents the semantics of LTL$_f$ in an intuitive manner with formulas constructed from atomic propositions *a, b*. LTL$_f$ formulas are on the left, while the circles indicate the states of the propositions, with x symbolising an arbitrary value.

2.4 Declare

As for the declarative language, we focus on DECLARE [23]. Unlike procedural models, where all allowed executions must be explicitly represented, DECLARE has an open flavour 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 the rules. Besides, it is grounded on a well-established

semantics: given a set of activities T, each DECLARE rule is a LTL$_f$ formula over T (with finite execution semantics) and the set of allowed finite executions are those satisfying the formulas.

Definition 7 (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 $t_1, t_2 \dots t_n$ such that $t_1, t_2 \dots t_n \vDash \Phi$.*

Name	LTL	Explanation
Absence	$\neg \diamond \mathsf{A}$	A must never occur.
Existence	$\diamond \mathsf{A}$	A must occur at least once.
Response	$\Box(\mathsf{A} \rightarrow \diamond \mathsf{B})$	If A is executed, then eventually B must be executed.
Chain response	$\Box(\mathsf{A} \rightarrow \bigcirc \mathsf{B})$	If A is executed, then B must be executed next.
Alt. response	$\Box(\mathsf{A} \rightarrow \bigcirc(\neg \mathsf{A} \mathcal{U} \mathsf{B}))$	When A occurs, it must be followed by B, without any A occurring in-between.
Alt. precedence	$(\neg \mathsf{B} \mathcal{W} \mathsf{A}) \wedge \Box(\mathsf{B} \rightarrow \bigcirc(\neg \mathsf{B} \mathcal{W} \mathsf{A}))$	When B occurs, it must have been preceeded by A, without any B occurring in-between.
Exclusive choice	$(\diamond \mathsf{A} \vee \diamond \mathsf{B}) \wedge \neg(\diamond \mathsf{A} \wedge \diamond \mathsf{B})$	A or B must occur, but not both.
Precedence	$\neg \mathsf{B} \mathcal{W} \mathsf{A}$	B can occur only when A has occurred.
Succession	$\Box(\mathsf{A} \rightarrow \diamond \mathsf{B}) \wedge (\neg \mathsf{B} \mathcal{W} \mathsf{A})$	B must occur after A and A must occur before B.

Table 1: Table of Declare rules with LTL$_f$ translations [7]

Table 1 shows a selection of the most common DECLARE templates with explanations and LTL$_f$ translations.

Using the activity names from Figure 1, we can define the following DECLARE rules:

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

Intuitively, these rules say that activity ask for updates has to occur before predict risk and check medical history. While, ask to recheck should not occur in the process execution.

2.5 Automaton

Generally, a *finite-state machine* (the term *automaton* is also used in some cases) is a construct showing changes of states based on a given input. While *regular languages* are recognized by finite-state machines. We can use finite-state machines and regular languages to formally express the behaviour contained in both procedural and declarative models. For that reason, we introduce finite-state machines, regular languages and the relationships between them.

Definition 8 (Nondeterministic finite-state machine)**.** *A nondeterministic finite-state machine is a tuple* $A_N = (S, \Sigma, \delta, s_0, F)$*, where* S *is a finite set of states,* Σ *is a finite set of symbols, called alphabet,* δ *is a transition function:* $\delta : S \times \Sigma \to \wp(S)$*,* $s_0 \in S$ *is the initial state and* $F \subseteq S$ *is the set of final states.*

Definition 9 (Deterministic finite-state machine)**.** *A deterministic 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:* $\delta : S \times \Sigma \to S$*,* $s_0 \in S$ *is the initial state and* $F \subseteq S$ *is the set of final states.*

As can be seen the difference between the deterministic finite-state machine (DFSM) and nondeterministic finite-state machine (NFSM) is that the transition function of the DFSM returns one state, while in case of the NFSM it returns a set. Intuitively the difference is that for DFSM given a state and an input there is a single next state, while in case of NFSM there might be zero or multiple possible next states.

Before we can define the languages of the finite-state machine, we need to define the execution path for the general finite state machine, be it deterministic or nondeterministic.

Definition 10 (FSM execution paths)**.** *Given a finite-state machine* A*, we define the set* Π *of paths* (s_0, σ_1, s_1)*,* (s_1, σ_2, s_2) *...* $(s_{n\text{-}1}, \sigma_n, s_n)$ *in* A*, where for each* $i \in \{1, \dots, n\}$*,* $s_i \in S$ *and* $\sigma_i \in \Sigma$ *inductively as follows:*

- *Base:* $(s_0, \sigma_1, s_1) \in \Phi$*, if* $(s_0, \sigma_1, s_1) \in \delta$*.*
- *Inductive: If* (s_0, σ_1, s_1)*,* (s_1, σ_2, s_2) *...* $(s_{n\text{-}1}, \sigma_n, s_n) \in \Pi$ *and* $(s_n, \sigma_1, s_{n+1}) \in \delta$ *then* (s_0, σ_1, s_1)*,* (s_1, σ_2, s_2) *...* $(s_n, \sigma_{n+1}, s_{n+1}) \in \Pi$

Definition 11 (FSM language)**.** *Given a finite-state machine* A *and a set of its paths* Π*, we define the language* $\mathcal{L}_A \subseteq \Sigma^*$ *as the smallest set such that:*

- *if* $s_0 \in F$ *then* $\varepsilon \in \mathcal{L}_A$ *and*
- *if* (s_0, σ_1, s_1)*,* (s_1, σ_2, s_2) *...* $(s_{n\text{-}1}, \sigma_n, s_n) \in \Pi$ *then* $(a_0, a_1, \dots, a_n) \in \mathcal{L}_A$*.*

In Definitions 10 and 11 we use the term finite-state machine to reference both deterministic and nondeterministic finite-state machines since the definitions are general enough to cover both cases.

In the literature, the word *automaton* is sometimes used 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 thesis, we use the words automaton and deterministic finite-state machine as synonyms.

If a language $\mathcal{L}$ is a $\mathcal{L}(A)$ for an automaton A, then $\mathcal{L}$ is called a *regular language* and A accepts language $\mathcal{L}$ [18].

It is important to note that every NFSM can be transformed into a DFSM accepting the same language. While an NFSM recognising a specific language can be represented using n states, it could take up to 2^n states to represent a DFSM of the same language.

Closure properties of regular languages tell us that regular languages are closed under certain operations, meaning that if one (or more) languages are regular then applying certain operations to them will result in languages that are also regular. Closure properties allow us to say that the complement of the language, union, intersection and difference of two languages are regular.

Suppose we have two automata $A_1 = (S_1, \Sigma, \delta_1, s_1, F_1)$ and $A_2 = (S_2, \Sigma, \delta_2, s_2, F_2)$ and we want to take a union, intersection or a difference of languages $\mathcal{L}_1$ and $\mathcal{L}_2$, which automatons A_1 and A_2 respectively accept, we need to construct a cross-product automaton of A_1 and A_2. The resulting automaton is $A_{CP} = (S_{CP}, \Sigma, \delta_{CP}, s_0, F_{CP})$, where:

- $S_{CP} = S_1 \times S_2$,
- $\delta_{CP} = ((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$,
- $s_0 = (s_1, s_2)$,

for every $p \in S_1$, $q \in S_2$, a $\in \Sigma$.

- A_{CP} accepts $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$, if $F_{CP} = \{(p, q) \mid p \in F_1 \vee q \in F_2\}$.
- A_{CP} accepts $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$, if $F_{CP} = \{(p, q) \mid p \in F_1 \wedge q \in F_2\}$.
- A_{CP} accepts $\mathcal{L} = \mathcal{L}_1 - \mathcal{L}_2$, if $F_{CP} = \{(p, q) \mid p \in F_1 \wedge q \notin F_2\}$.

In [10] authors show a way to transform LTL_f into an automata. The result is achieved by transforming a LTL_f formula ϕ into a LDL_f formula ϕ'. LDL_f is the combination of LTL and regular expressions based on finite traces. Regular expressions provide a declarative way of defining a language that an automaton accepts

and are more expressive that LTL_f [18]. Hence, also LTL_f formulas can be represented using automata. Every LDL_f formula is transformed into an alternating automaton on words (AFW) which accepts a language consisting of traces making ϕ' true. Since AFW is a specific case of NFSM, it can be converted into NFSM which in turn can be transformed into a DFSM. Figure 3 shows an automaton generated from a set of DECLARE rules described in (1).

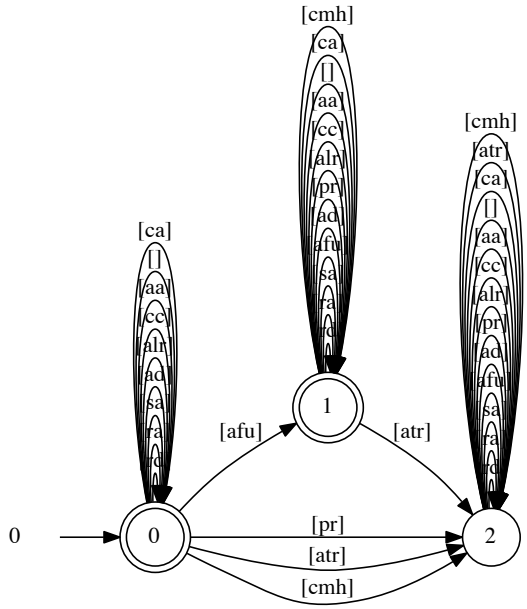

Figure 3: Automaton generated from LTL_f formulae

3 Related Work

The main ideas behind the topic of this thesis are the integration of procedural and declarative models and the repair of procedural models. To understand the state of the art behind these ideas, the following research questions were asked:

- How can declerative and procedural process models be integrated?
- What are the main methods for repairing a procedural process model?

The following key phrases were used to search the databases for relevant articles: "Model repair", "Procedural and Declarative models", "Model checking". In the next sections we report the findings.

3.1 Procedural vs declarative

Recent research has presented evidence about the synergies between imperative and declarative approaches [27]. Reijers et al. [27] conducted a workshop in order to find out whether declarative process modelling techniques could be used in practice. In the opinion of the participants of the workshop, declarative languages inherently provide a higher level of abstraction than the procedural ones, so they can be used to quickly change models by adding and removing constraints, without redoing the entire models. Yet, declarative languages are not suitable for well-structured processes, which can be easily modelled using procedural languages. In [24], Pichler et al. investigate procedural and declarative modeling languages w.r.t. the model understanding and conclude that procedural models are more understandable, though they concede that it might partially come from the inexperience of the participants with the declarative languages. While declarative languages are not as well-known as procedural ones, they do have certain advatages, as shown in [27], therefore an area of interest are the hybrid languages which combine the procedural and declarative languages.

Recently, several hybrid process modeling notations have been proposed. In particular, De Giacomo et al. [8] propose a conservative extension of BPMN for declarative process modeling, namely BPMN-D, and show that DECLARE models can be transformed into readable BPMN-D models. BPMN-D is a conservative extension since it only adds constructs to BPMN, meaning that every BPMN-D model can be transformed into a BPMN model. The translation of a DECLARE model into a BPMN-D model consists of two stages. Firstly, the declarative model is transformed into a constraint automaton, which is a declaratively and more concisely labelled finite-state machine (FSA) and, in the second stage, that automaton is translated into a BPMN-D model. To translate the constraint automaton into a BPMN-D model, the authors provide an algorithm that maps the automaton

states into BPMN-D states and sequence flows, and automaton transitions into BPMN-D transitions taking into account the constrains of the automaton.

Westergaard and Slaats [34] aim at combining declarative languages DECLARE and DCR graphs with a procedural language, Coloured Petri nets, to get an integrated approach, that combines positive aspects of both paradigms. The main idea behind their approach is to identify transitions of the Petri net, tasks of the DECLARE model and events of DCR graph models and then add places and arcs from Petri nets, constraints from DECLARE and relations from DCR graphs to constrain the resulting model. An execution is considered to be accepting only if all of the underlying models accept it. Additionally, they provide a step-wise semantics to simulate their combined model. A recent implementation of this technique is made available in CPN Tools 4.0 [33].

De Smedt et al. [11] extend the work in [34] by defining a semantics based on mapping DECLARE rules to Petri nets with Reset and Inhibitor Arcs. Additionally, the authors investigate how difficult it is to combine different DECLARE templates with the procedural model. Lastly, they provide modelling guidelines for combining the procedural and declarative constructs.

In [28], Slaats et al. present a formal semantics for a hybrid process modeling notation. In their proposal, a hybrid process model is hierarchical, and each of its sub-processes may be specified in either a procedural or declarative fashion. In [21], Maggi et al. propose an algorithm for discoverying hybrid models from event logs. The main idea behind their algorithm is to divide the log into structured and unstructured groups of events and then use procedural mining techniques on structured groups to discover procedural sub-processes and declarative mining techniques on unstructured groups for discovering declarative sub-processes. Then a top-level process is mined which allows combining the discovered procedural and declarative sub-processes into a hybrid model.

3.2 Model repair

In [15], Fahland and van der Aalst addressed the problem of repairing procedural process models based on the information retrieved from an event log. They aimed at keeping the repaired model as similar as possible to the original model, while capturing all the possible behaviors from the event log. The authors identified the minimum amount of changes needed to replay the event log on the model by aligning the two. Once the deviations have been identified the repair algorithm consists of three parts - firstly, adding structured loops, secondly, adding sub-processes and thirdly, removing unused nodes. Using these techniques resulted in repaired models more similar to the original models than the ones achieved through applying process discovery algorithms to the event logs.

Buijs et al. [4] use similarity to the original model to choose the most suitable

model out of a set of candidate models mined from the event log. Since they are rediscovering the model, they are not modifying the original model.

In [16], Gambini et al. provide an algorithm that takes as input an unsound process model and returns a set of alternative models that contain less behavioral errors than the original one. The technique is based on Multi-Objective Simulated Annealing (MOSA). MOSA provides many advantages over other optimization algorithms with the main one being that it does not get stuck in the local optimum and can be used to provide a diverse set of resulting models.

In [1], Armas-Cervantes et al. argue that the log-based model repair approaches suffer from an important limitation by adding too much behaviour to the model and producing models that are over-generalised. They suggest a novel iterative method that provides user with a graphical representation of the repair operations with the highest impact. Differences between the model and the log are identified by transforming both into graphs of events, with nodes being linked by the relations between the events. Using the syncronised product of the two graphs deviations between the log and the model are mapped into patterns which are then used for visualising the problems for the users, who can manually repair them. Compared to [15] their approach returns models that are still similar to the original but are not as over-generalised.

Another source for repairing a model could be the model itself, which is investigated in a paper written by Lohmann and Fahland [20]. This can be done using model checkers. Model checkers can check properties of process models expressed in terms of temporal logic. In case of business processes the most well-known property is soundness, which encompasses the lack of dead-locks, live-locks, dead code and that the model is terminated correctly. Lohmann and Fahland investigate the error paths of model checkers. They propose a way of reducing error paths of model checkers by focusing on choices made during executions and removing unnecessary information.

3.3 Discussion

Based on the literature we have examined we say that there is a real need for integrating the procedural and declarative languages. The need does not stem only from academia, the industry is interested as well [27]. For that reason, the researchers have been looking at hybrid languages, which manifest in two different ways. One option is to create a new mixed language comprised of the best features of procedural and declarative (e.g. [8]), the other way is to combine them while still keeping the specifications separate (e.g. [28, 21]).

Model repair has mostly been attempted by using the event logs to repair the procedural process models [15, 1]. Additionally, there have been efforts to address the behavioural and syntactical errors in the model [16, 20].

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 thesis, we approach the problem from a logic-based perspective and propose two initial solutions to it.

4 Problem & Approach

In this section, we will introduce the problem at hand and describe our approach to solving it.

4.1 Problem

Suppose we have a bank called Banco BPM (BBPM) with 100 branches in Italy, Spain and Germany and a headquarters in Milan, Italy. It has assets of about 25 billion, equity 2 billion and profit of 100 million euros. As all other banks in Europe, BBPM is a subject to government and European Union regulations. Additionally, bank management can also make policy changes concerning the way business is done. Since BBPM is a moderately big bank it has a lot of processes to manage and whenever there are law or regulation changes affecting its processes business analysts at BBPM have to manually identify the affected processes and change the models to reflect the new reality. Since this is a complex and error-prone work, the analysts are interested in a way of doing it either semi- or fully automatically.

To elaborate, the problem we face is how do we integrate the reality of the declarative business rules into procedural process models in a way that the end result would preserve key properties (such as the similarity to the original model) making it useful for the business analysts.

4.2 Approach

For the procedural process models, we use Petri nets while for declarative business rules we use DECLARE. We aim to combine the two into a combined model that we want to represent using only procedural notation. This is useful for two reasons, firstly, most of the workflow systems currently use procedural notations and it would not be feasible to start changing them to take into account the declarative specification, secondly, people are more used to the procedural languages, since they have been used for a longer time and they are easier to comprehend.

We focus only on the cases where the procedural and the declarative models do not conflict with each other. While this is, of course, a real possibility in the real world, analysing the root causes of the clashes would require substantially different techniques than the ones we currently propose.

We propose two techniques for dealing with consistent combined models, i.e. when the procedural and the declarative part of the combined model do not conflict. We assume that the designers of the original models created them for specific reasons, so we aim to keep the adopted model similar to the original and not add any extra behaviour. Since the proposed techniques can return multiple results,

we had to come up with a set of heuristics and metrics to choose the most suitable result.

5 Framework

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

Definition 12 (combined model)**.** *Given a language of a workflow net $\mathcal{L}_W$ and a language $\mathcal{L}_D$ which is compliant with Φ. A combined model is a couple $C = (W, \Phi)$ 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 compose 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 the* ask for updates *is performed. Such guidelines are defined, using the* DECLARE *rules, in* (1)*, in Section 2.4. 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 Figure 4a and 4b 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 4a is the case of Example 1. Indeed, the execution

$\pi_1 = (\mathsf{receive\ application}, \mathsf{check\ application}, \mathsf{ask\ for\ updates}, \mathsf{retrieve\ data}, \mathsf{analyse\ data},$
$\mathsf{assess\ loan\ risk}, \mathsf{check\ medical\ history}, \mathsf{check\ career}, \mathsf{predict\ risk}, \mathsf{send\ form})$

satisfies both the WF-net in Figure 1 and the three declarative rules in (1), hence $\pi_1 \in \mathcal{L}_{W \cap D}$. While execution

$\pi_2 = (\mathsf{receive\ application}, \mathsf{check\ application}, \mathsf{ask\ to\ recheck}, \mathsf{retrieve\ data}, \mathsf{analyse\ data},$
$\mathsf{assess\ loan\ risk}, \mathsf{check\ medical\ history}, \mathsf{check\ career}, \mathsf{predict\ risk}, \mathsf{send\ form})$

satisfies the WF-net in Figure 1 but not the rules in (1), meaning $\pi_2 \in \mathcal{L}_W \backslash \neg\mathcal{L}_D$. Execution

$$\pi_3 = (\mathsf{ask\ for\ updates}, \mathsf{predict\ risk})$$

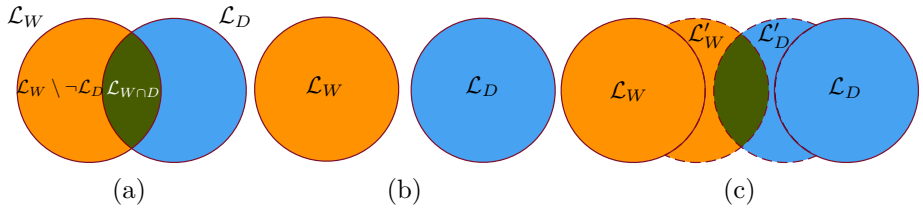

Figure 4: Graphical representation of the relationships between $\mathcal{L}_W$ and $\mathcal{L}_D$.

would satisfy the rules but not the WF-net and $\pi_3 \in \mathcal{L}_D \setminus \neg\mathcal{L}_W$.

In Figure 4b, all W executions do not conform to the 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 4c. 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 a separate problem and we focus on the "core" case of how to select a nonempty set of net executions compliant with the declarative rules.

Given the above, let us assume the language $\mathcal{L}_{W\cap D}$ of $C = (W, \Phi)$ is not empty. The problem we address is the following: *ideally* we want to find and return a model W' such that the set of accepted execution is $\mathcal{L}_{W'} = \mathcal{L}_{W\cap D}$. For (theoretical and practical) reasons that will be clear in the 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 5 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 model not compliant with the declarative model $\alpha = \mathcal{L}_W \cap \neg\mathcal{L}_D \cap \mathcal{L}_{W'}$, 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 and not accepted by the procedural $\delta = \neg\mathcal{L}_W \cap \neg\mathcal{L}_D \cap \mathcal{L}_{W'}$.

When we adapt model W to W', we are looking for a model that would allow only for the behaviour described by the language $\mathcal{L}_{W\cap D}$. This means that apart from not adding new behaviour, W' should lose minimal amount of behaviour (desideratum *D1*). Another key aspect of the repair process is the structural similarity of W and W'. By structural similarity we mean the (graph edit) distance between W and W', as well as the capability of not altering, in W', the relevant properties characterising the original model W, such as block-structurdness or the activity duplication (desideratum *D2*). Unfortunately, behavioural and structural

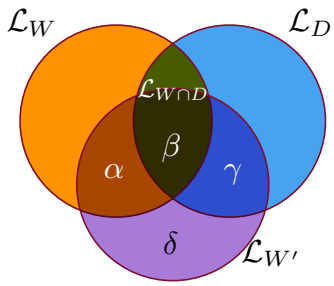

Figure 5: The relationships of languages $\mathcal{L}_W$, $\mathcal{L}_D$ and $\mathcal{L}_{W'}$.

similarity tend to be conflicting requirements. The more behaviour of W' we are able to replay (e.g. the whole $\mathcal{L}_{W \cap D}$), more it starts diverging from W and lose other structural properties, the less understandable it is for the business analysts. Similarly, the more similar W' is to the original W (i.e. the more understandable W' is), the more likely it is that W' has lost behaviour that was in $\mathcal{L}_{W \cap D}$ or behaviour has been added that was not it $\mathcal{L}_{W \cap D}$.

6 Contribution

Given a combined model $C = (W, \Phi)$, the problem we want to address is to find a WF-net W' such that the set of accepted executions $\mathcal{L}_{W'}$ is as close as possible to $\mathcal{L}_{W \cap D}$.

This problem is tackled in two separate steps: (1) first, $\mathcal{L}_{W \cap D}$ is computed, and then (2) the WF-net is found. We exploit the well-known equivalence between (regular) languages and automata and propose two automata-based approaches to solve both steps.

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., [18], briefly summarised in Subsection 2.5) 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 the 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. Figure 6 shows a fragment of a reachability graph[1] of the procedural model described in Example 1. While Figure 7 shows A_W of Example 1. While the essence of Figures 6 and 7 is the same, the representation is different, since Figure 6 shows the markings explicitly.

Definition 13 (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 the automaton A_D from Φ by exploiting the results and algorithms in [6], which guarantees that $\mathcal{L}(A_D) = \mathcal{L}_D$. Finally, we compute the intersection as the synchronous product [18] of A_W and A_D.

[1] Generated using WoPeD, available at: `http://woped.dhbw-karlsruhe.de/woped/`

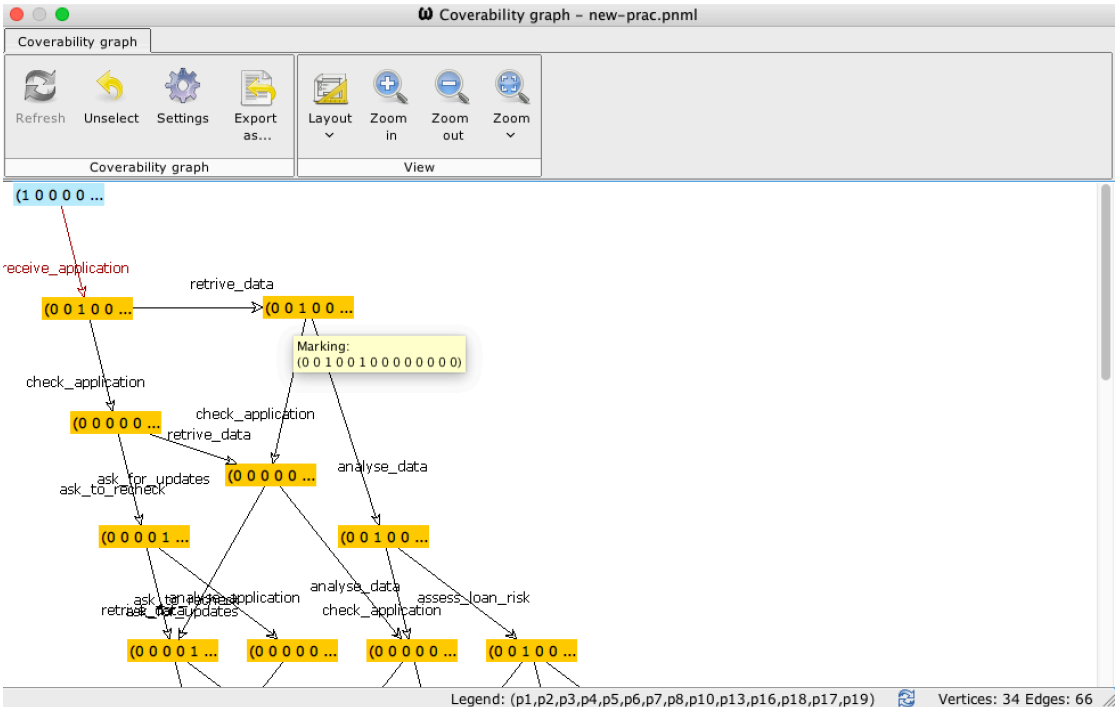

Figure 6: Fragment of the reachability graph of the running example.

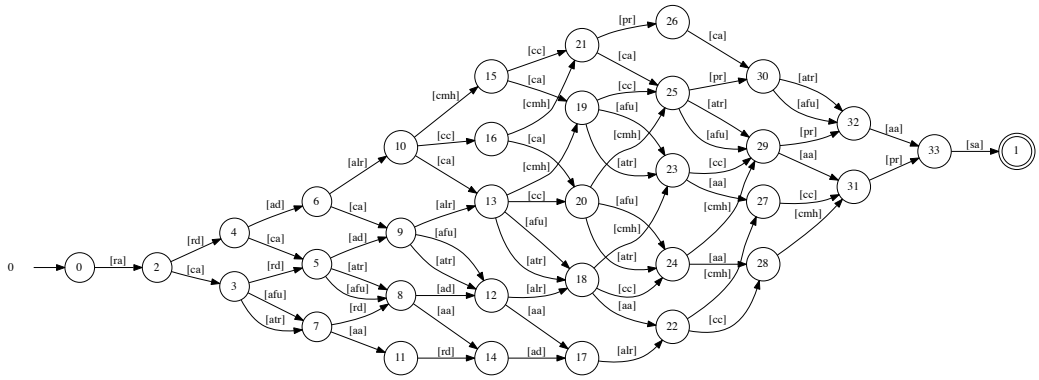

Figure 7: Automaton A_W of Example 1

An intersection of two automatons is defined in Section 2.5 but in order to proceed we need an intersection of two automata that keeps the information about the markings of WF-net W.

Definition 14 (Automaton intersection with markings). *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.*

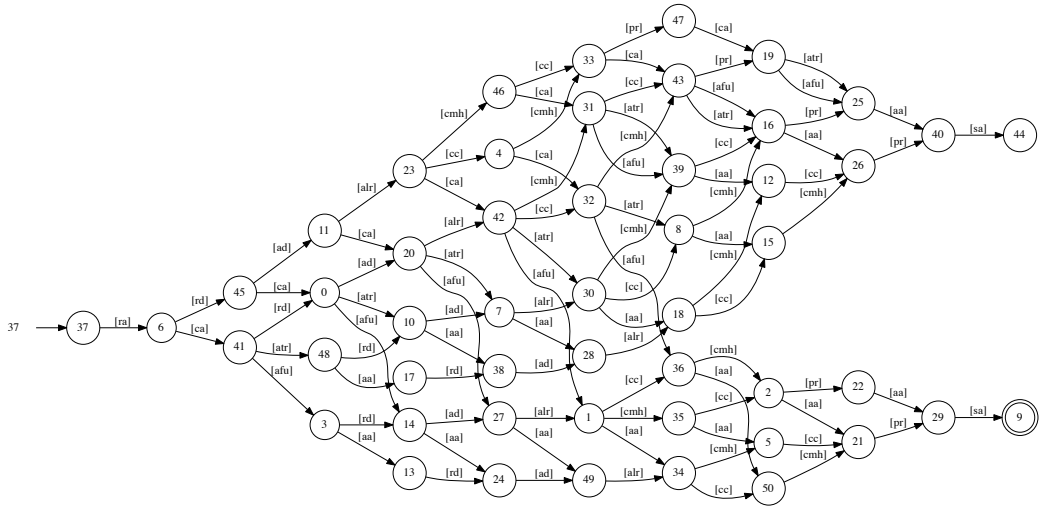

Figure 8: Automaton $A_{W \cap D}$ of Example 1

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 $A_{W \cap D}$. Figure 8 shows $A_{W \cap D}$ generated as an intersection of the automata based on procedural (Figure 7) and declarative models (Figure 3).

In order to accommodate different needs of W', described in the end of Chapter 5, we propose two approaches to solve the problem, one suited to address desideratum *D1*, language similarity (see Section 6.1), and the other suited to address desideratum *D2*, structural similarity (see Section 6.2).

6.1 Petri net synthesis

The basic idea of the PN synthesis [2] is to decide whether a given automaton is isomorphic to the reachability graph of some Petri net. Intuitively, isomorphism means that two automata are the same apart from the node labels. While there is a finite amount of such nets, checking all of them is not a viable solution. Another

way is to synthesise a candidate net from the automaton and check whteher its reachability graph is isomorphic to the automaton. 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.

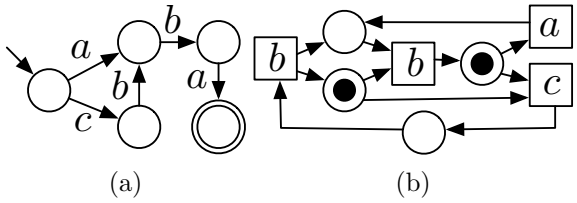

Figure 9: Simple automaton and its complex synthesized Petri net.

Because of 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 9a, where the leftmost state is the initial one and the double-circled one is the final, the synthesized net[2] in Figure 9b 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.

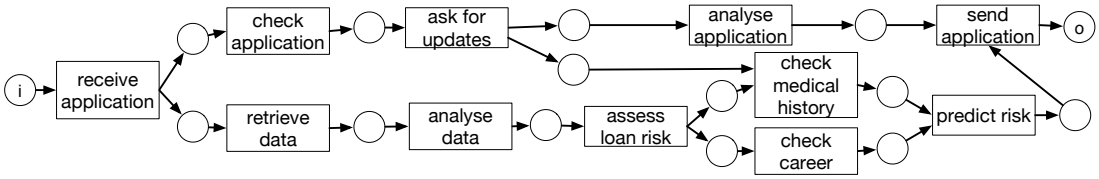

Figure 10: WF-net resulting from the application of Petrify to the net and rules in Example 1.

Because of the formal specification of the problem we are able to successfully able to employ PN synthesis. Figure 10 shows a Petri net resulting in applying Petrify to the net and the rules in Example 1. Notice that the resulting net is no longer block-structured as the original was. The violation of the block-structuredness comes after the activitiy ask for updates. So while we are able to achive desideratum *D1*, we are not able to achive *D2*.

[2]Obtained using petrify tool, available at: `http://www.cs.upc.edu/~jordicf/petrify/`

6.2 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)* does not have duplicate activities and *(iii)* is block-structured. In order to fulfill the above requirements, we are willing to possibly sacrifice on language similarity.

We started with an understanding that we can use the automaton as a source of information for the repair. Since we have met the hypothesis of the non-empty intersection of the procedural and the declarative models, intuitively, all that has to be done is the removal of exclusive branches and/or possible interleavings of the parallel branches from the original net. We tried to use the automaton to identify what part of the net causes non-conformance to the rules. We arrived at an initial algorithm that is the last step ($flatten$) of the current algorithm. Next, we tried to figure out ways of improving the algorithm, in order to lose as little behaviour as possible, and arrived at the current version of the algorithm.

The main idea behind this approach is starting from the original net W and try to remove the behaviours 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.

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, visible in Figure 8.

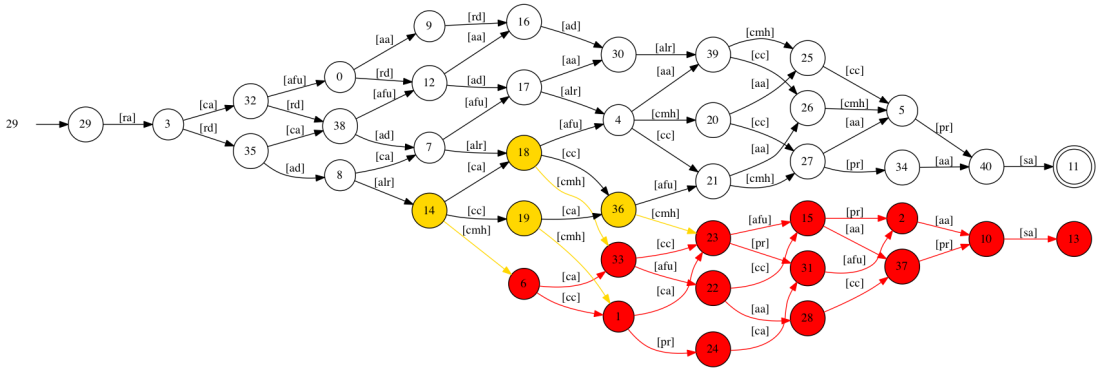

Figure 11: Automaton $A_{W \cap D}$ with semi-bad and bad states.

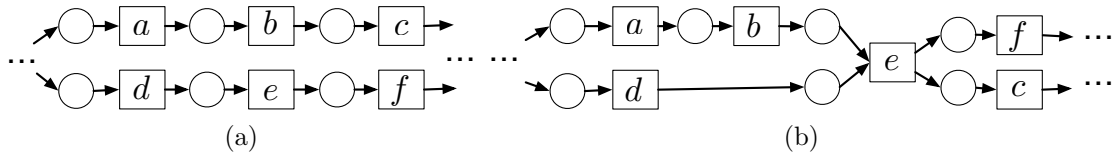

Figure 12: Original net (a) and modified one (b) to accomodate *precedence*(b, e) with synchronization points.

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 removing branches of exclusive choices that do not comply with Φ. In the running example, presented in Figure 1, ask to recheck is removed.

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 the 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*. On Figure 11 these states are coloured red. 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: $B = \{s \in (S_{W\cap D} \backslash F_{W\cap D}) \mid \forall s'.(s' \in F_{W\cap D} \rightarrow (s, s') \notin R)\}$. Also, we define a *semi-bad* state $SB = \{s \in (S_{W\cap D} \backslash B) \mid \exists s', a.(s' \in B \wedge (s, a, s') \in \delta_{W\cap D})\}$. On Figure 11 semi-bad states are coloured yellow. 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: $PS_a = \{s \in SB \mid \forall s, \exists a.(s, a, s') \in \delta_{W\cap D} \wedge s' \in B\}$. In the running example we can see one problem set, identified by the activity check medical history. It consists of states numbered (Figure 11) 14, 18, 19, 36.

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

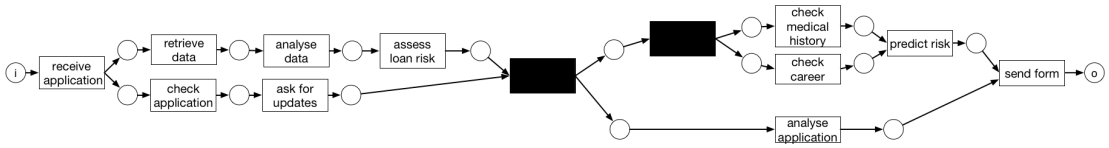

Figure 13: WF-net resulting from the application of ARNE to the net and rules in Example 1.

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 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 12 graphically shows how *addSyncPoints* works when dealing with the DECLARE rule $precedence(b, e)$. In practice, finding synchronization points works as follows: firstly, we find a set of states belonging to a problem set, from which it is not possible to reach any other state of the same problem set. We call this set the *last states* of the problem set LS. Formally, $LS = \{s \in PS_a \mid \forall s, \forall b.(s, b, s') \in \delta_{W \cap D} \wedge s' \notin PS_a\}$. Intuitively, each state in LS has at least two outgoing arcs, one to a bad state, another to a good state. We take one of the states from LS and look at the transitions leading to a good and a bad state. The transition leading to a good state will become a source of the synchronization, while the transition leading to the bad state will become the target. In Figure 11 the state belonging to LS is 36. An arc going to a good state is labelled with ask for updates, while the one to the bad state with check medical history. If we attempted to synchronize using these two transitions, we would get a non-block-structured model. To keep the model block-structured we have to synchronize in PN branches that are on the same nesting level. In the running example, the first such transition is assess loan risk. Yet in general, if we synchronize on an AND-split we would not get a block-structured model. An alternative is to use hidden transitions. In this case we would have to add two hidden transitions, as visible on Figure 13, which will allow us to keep block-structuredness and keep more behaviour than other alternatives, such as synchronizing before the AND-split.

5 - flattening. When, after adding synchronization points, the model still has non-compliant behaviors and automaton does not provide enough information for additional synchronization points, function *flatten* is used, which essentially "flats" the parallelism by concatenating parallel branches. To explain flattening, we first introduce the concept of a *semi-bad front*. First level semi-bad front is a set of semi-bad states from which it is possible to reach in one step a bad state. 2-nd

level semi-bad fronts are a set of semi-bad states from which it is possible to reach a n-1 semi-bad front in one step. Let the first front be denoted as $SB_1 = SB$. In this case $SB_n = \{s \in (S \backslash SB_{n-1}) \,|\, \exists s', \exists a.(s' \in SB_{n-1} \wedge (s, a, s') \in \delta_{W \cap D})\}$. We apply the *flattening* on each semi-bad front until the model is complient with the rules.

The essence of the *flattening* function is finding places in one branch of the net to connect them to the another branch of the net. We do this using the pairs of semi-bad and bad states. We define the pair as $\overline{S} \subseteq SB \times B = \{s_1, s_2 \,|\, \exists a.(s_1, a, s_2) \in \delta_{W \cap D}\}$. Additionally, we would like to define a function ct that takes a pair of semi-bad and bad states and outputs a transition connecting them. $ct : SB \times B \to \wp(T)$. $ct(sb, b) = \{a \mid \exists a.(sb, a, b) \in \delta_{W \cap D}\}$. Given an automaton $A_{W \cap D}$ as defined in Definition 14 and a state $s \in S_{W \cap D}$ we define inductively a set of paths $\Pi(s, a)$ starting from s as the set such that:

- $s \in \Pi(s, a)$
- if $s \overset{a_0}{\to} s_1 \overset{a_1}{\to} ... \overset{a_{n\,1}}{\to} s_n \in \Pi(s, a)$ then $\forall s_{n+1}, a_n.(s_n, a_n, s_{n+1}) \in \delta$ implies $(s \overset{a_0}{\to} s_1 \overset{a_1}{\to} ... \overset{a_{n\,1}}{\to} s_n \overset{a_n}{\to} s_{n+1}) \in \Pi(s, a)$

Given $\Pi(sb, a)$ and $a \in ct(sb)$ we define a set of paths: $\Delta(sb, a) \subseteq \Pi(sb, a) \mid \pi \in \Delta(sb, a)$ implies:

- π is maximal.
- a does not belong to π.

Given $\Delta(sb, a)$ we define the set $ES_{\Delta(sb,a)}$ of all ending states of paths, not containing sink states, in $\pi(sb, a)$. Because of maximality, we are sure that $\forall s \in ES_{\Delta(sb,a)} \exists s' \mid (s, a, s') \in \delta$. Now we have a state s that represents a marking in the net, which intuitively is the last marking before the problem occurs and a rule becomes violated. Using s and s' we take the difference of the marking using the function lp with a signature $lp : ES_{\Delta(sb,a)} \times S \to \wp(P)$, while being defined as follows: $lp(s, s') = \{p \mid SM(s)(p) = 1 \wedge SM(s')(p) = 0\}$. We can use the places returned by the lp to connect them to the transition that causes the problem, essentially flattening a part of the net.

Algorithm 1 shows compactly the five steps described above.

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. In case of rules containing *next* operator is far more difficult to extract information from the automaton concerning the location of the problem than it is for the other

Algorithm 1: ARNE algorithm

Data: W + DECLARE rules
Result: Repaired W'

```
1  A_{W∩D} ←intersect(W, Φ);
2  W' ←removeUnusedPlaces(A_W, A_{W∩D});
3  if W' is not compliant with DECLARE rules then
4      set of PS_a ←getProblemSets(A_{W∩D});
5      while set of PS_a is not empty do
6          LS ←getLastStateOfTheProblemSet(PS_a);
7          for ls ∈ LS do
8              getSyncSourceAndTarget();
9              sync();
10             checkComplianceToRules();
11     if W' is not compliant with DECLARE rules then
12         set of SB ←getFronts();
13         while set of SB is not empty do
14             S̄ ←getSemiBadBadPairs();
15             for SB × B ∈ S̄ do
16                 ES_{Δ(sb,a)} ← getPaths(SB, ct(SB × B));
17                 getSourceAndTarget();
18                 flatten();
19                 checkComplianceToRules();
```

types of rules. For that reason, other heuristics might be necessary for the rules with the *next* operator. As for the cycles, when combined with declarative rules, they may lead to ambiguous meaning. 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. It can be confusing how many times can the loop be passed and a executed and should b be executed before each loop pass. Because of this, futher investigations are required on how to handle cycles in the procedural models. One possible way would be to limit, with a threshold k, the number of times the loops could be passed through.

We close the section by showing the graphical representation of the WF-net resulting by applying ARNE to Example 1 (Figure 13). It should be noted that in the running example only the first four steps were taken, as the *flattening* was not needed.

7 Implementation

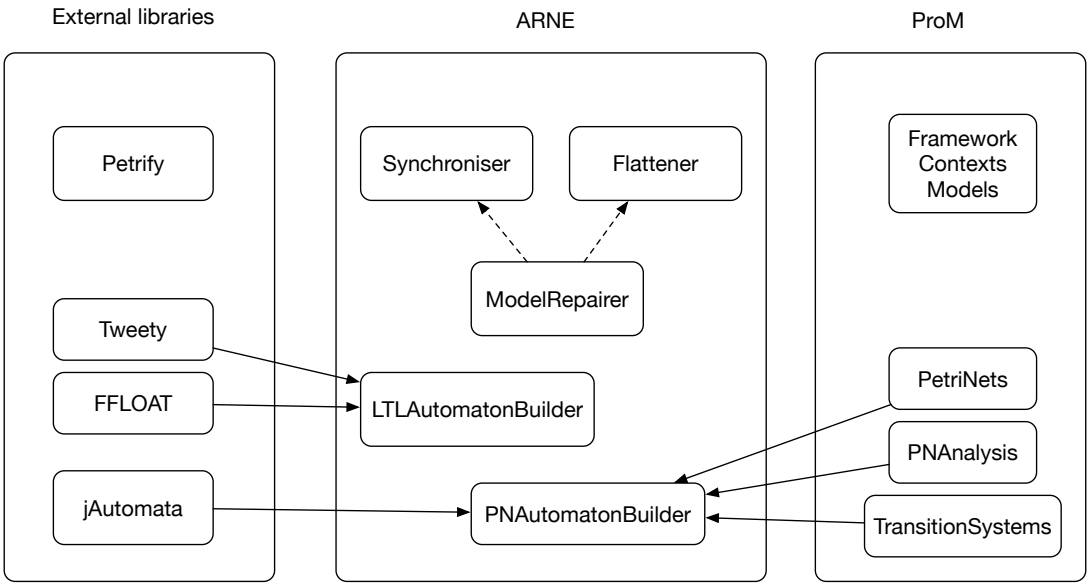

Figure 14: The architecture of ARNE *ProM* plug-in.

The two automata-based approaches, namely PN synthesis and ARNE, described in Sections 6.1 and 6.2 respectively, were implemented in a novel *ProM* plug-in.[3] For the PN synthesis, we incorporated the existing *petrify* tool.[4] The ARNE algorithm detailed in Section 6.2 was implemented from scratch and it makes use of the FLLOAT library[5] for the automata generation which implements the algorithm in [6]. Figure 14 presents an overview of the system architecture of the ARNE *ProM* plug-in. We use external libraries (s.t. jAutomata, FLLOAT, Tweety) and *ProM* libraries (s.t. PetriNets, PNAnalysis, TransitionSystems) to generate automata. Other major components (s.t. ModelRepairer) are used for PN modifications. FLLOAT library was implemented using Java 8, while the *ProM* online version only supports Java versions up to 7. Because of the incompatibility of different versions of Java, it is not possible to distribute the plug-in through the *ProM* package manager, but it can be run locally.

To convert a given Petri net into an automaton, code from the *ProM* plug-in *PNAnalysis*[6] was used.

[3] `https://github.com/alaponin/AutomatedRuletoNetEnactorProMPlugin`

[4] `http://www.cs.upc.edu/~jordicf/petrify/`

[5] `https://github.com/RiccardoDeMasellis/FLLOAT`

[6] `https://svn.win.tue.nl/repos/prom/Packages/PNAnalysis/Trunk/src/org/processmining/plugins/petrinet/behavioralanalysis/TSGenerator.java`

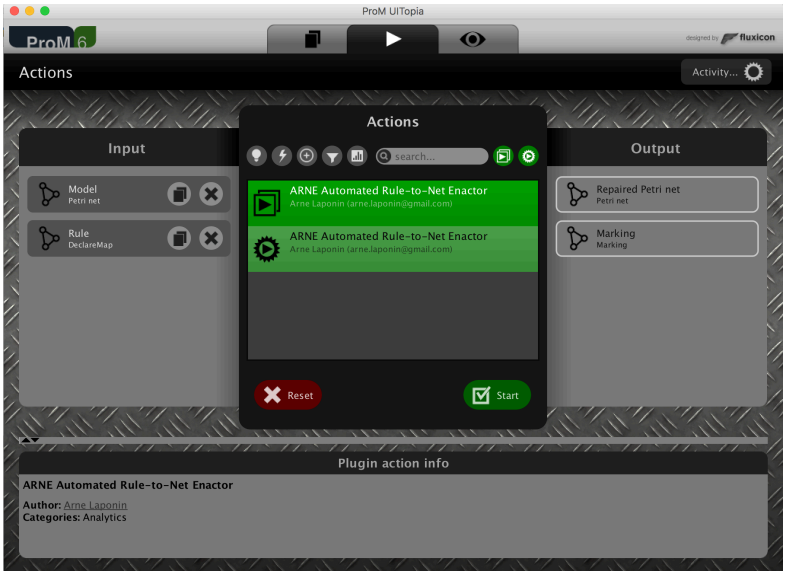

Figure 15: Input window of ARNE *ProM* plug-in.

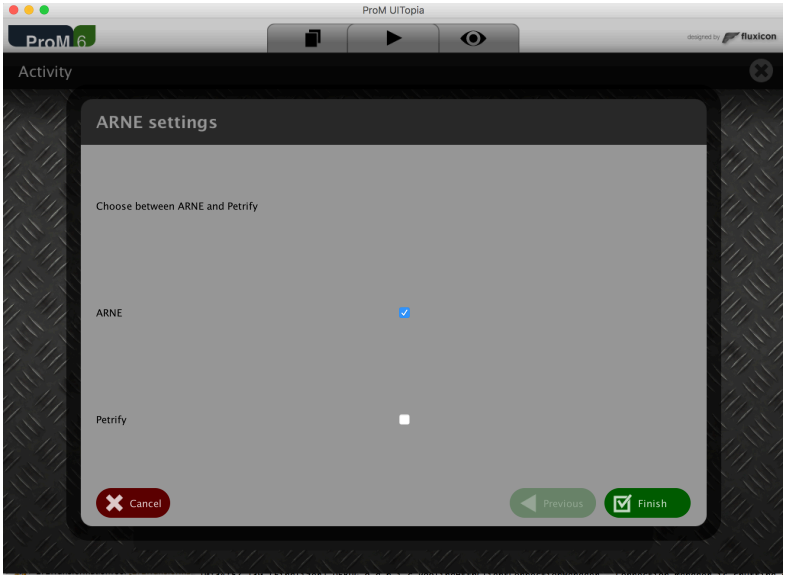

Figure 16: Dialog window of ARNE *ProM* plug-in.

As shown in Figure 15, the plug-in takes as inputs a PN and a set of DECLARE rules. Next, the user is prompted with the possibility to choose between using ARNE or *petrify* (Figure 16), based on whether she wants to preserve syntactical similarity or keep all behaviors of the original model. The plug-in returns a repaired process model as a PN (Figure 17). All repair operations presented in Tables 4

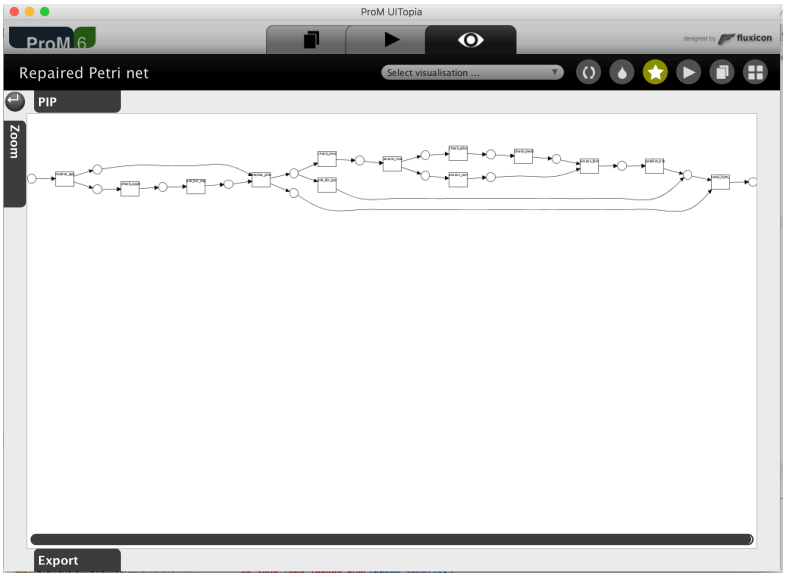

Figure 17: Results window of ARNE *ProM* plug-in.

and 5 took around couple of seconds on a standard laptop.

	# 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 2: Information about the procedural models

8 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 only belong to the set of desired behaviors β, thus solving the problem as stated in Section 6?

Q2: are the techniques only in β able to satisfy the other problem desiderata (*D1* and *D2* mentioned in Section 5), i.e., similarity to the original procedural net, block-structuredness and no duplicate activities?

8.1 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.

Table 2 illustrates some characteristics of the procedural models we use, while Figures 18, 19, 20 (see Appendix I) provide images of said models, with the activity names shortened for clarity. $W1$ is characterised by several (mutually exclusive) alternative branches, $W2$ is characterised by a high parallelism degree and finally, the distinctive feature of $W3$ is the presence of a loop. These distinctive features are added to the process names (as in Table 2), 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. In Table 3 we report the rules used for the process models. The prefix in the set name makes clear to which process the rules refer to.

Thus, our dataset is composed of 11 entries, one for each suitable combination of the three procedural models in Table 2 with the declarative rules above. For

Name	DECLARE template	LTL
$W1$-$D1$	*alt. response, precedence*	$(\Box(g \rightarrow (\circ(\neg g \,\mathcal{U}\, h)))) \wedge (\neg r \,\mathcal{W}\, s)$
$W1$-$D2$	*alt. precedence, response*	$((\neg g \,\mathcal{W}\, h) \wedge (\Box(g \rightarrow \circ(\neg g \,\mathcal{W}\, h)))) \wedge (\Box(ag \rightarrow \diamond ah))$
$W1$-$D3$	*alt. precedence, response, absence*	$((\neg ae \,\mathcal{W}\, c) \wedge (\Box(ae \rightarrow \circ(\neg ae \,\mathcal{W}\, c)))) \wedge (\Box(h \rightarrow \diamond g)) \wedge (\neg \diamond y)$
$W1$-$D4$	*precedence, response, response*	$(\neg ae \,\mathcal{W}\, c) \wedge (\Box(h \rightarrow \diamond g)) \wedge (\Box(r \rightarrow \diamond s))$
$W2$-$D1$	*response, response*	$(\Box(d \rightarrow \diamond o)) \wedge (\Box(k \rightarrow \diamond s))$
$W2$-$D2$	*response, precedence, absence*	$(\Box(g \rightarrow \diamond q)) \wedge (\neg t \,\mathcal{W}\, m) \wedge (\neg \diamond l)$
$W2$-$D3$	*response, absence, precedence*	$(\Box(g \rightarrow \diamond o)) \wedge (\neg \diamond e) \wedge (\neg m \,\mathcal{W}\, v)$
$W2$-$D4$	*response, alt. response, precedence*	$(\Box(k \rightarrow \diamond s) \wedge (\Box(d \rightarrow \circ(\neg d \,\mathcal{U}\, r))) \wedge (\neg g \,\mathcal{W}\, o)$
$W3$-$D1$	*precedence*	$\neg f \,\mathcal{W}\, h$
$W3$-$D2$	*response*	$\Box(k \rightarrow \diamond j)$
$W3$-$D3$	*alt. response*	$\Box(g \rightarrow (\circ(\neg g \,\mathcal{U}\, b)))$

Table 3: Evaluation rules

instance, $W1$ has been used in four datasets, one for each pair from $W1$-$D1$ to $W1$-$D4$ of rules. Similarly for the others.

Log-based technique - discovery. The main idea is to select 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. This is done by converting D into an automaton [32] and then checking which traces in Π_W are accepted by said automaton. Intuitively, $\Pi_W \cap \mathcal{L}_D$ corresponds to a random set of traces compliant with both W and D (and thus belonging to the intersection β in Figure 5). We then discovered the procedural W_{dscvr} model from log $\Pi_W \cap \mathcal{L}_D$ using the Heuristic miner [31].

Log-based technique - repair. We built the second log-based technique on the work done by De Giacomo et al. [9]. The main problem of their work is taking a event log trace t, aligning it to a DECLARE rule ϕ and finding a trace t' that satisfies ϕ. Trace t is transformed into a *trace automaton* T, while ϕ is converted into a *constraint automaton* A. In order to find t', A is augmented into A' that accepts t'. Using the described technique we can take the set Π_W, align its traces to the declarative rules in D and get the set Π_D that is complient to D. We then use the Repair Model (remove unused parts) ProM plug-in [15], described in Section 3.2, to repair W w.r.t. Π_D, thus obtaining a repaired W_{rpr}.

Procedure. 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 [5];
- using afromentioned log-based discovery technique on Π_W we discovered the model W_{dscvr}.

- starting from the set Π_W generated previously, we used the log-based repair technique, described above, to obtain a repaired W_{rpr};
- for each returned model $W_{new} \in \{W_{petrify}, W_{\text{ARNE}}, W_{dscvr}, W_{rpr}\}$, 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 w.r.t 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 [13], 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 2): $\alpha = \mathcal{L}(A_W \wedge \neg A_D \wedge A_{W_{new}})$, $\beta = \mathcal{L}(A_W \wedge A_D \wedge A_{W_{new}})$, $\gamma = \mathcal{L}(\neg A_W \wedge A_D \wedge A_{W_{new}})$ 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). More precisely:

- 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 $\mathcal{L}(A_\alpha) = \alpha = \emptyset$ and the boolean α is set to *true*.

- 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):

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

  where $\mid A \mid$ 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.

	W_{dscvr}					W_{rpr}				
Name	**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
$W1$-$D2$	Yes	No	0.57	0.86	T/F/T	Yes	No	0.85	1.00	F/T/T
$W1$-$D3$	Yes	No	0.58	1.00	T/F/F	Yes	No	0.85	1.00	F/T/T
$W1$-$D4$	Yes	No	0.57	1.00	T/F/T	Yes	No	0.84	1.00	F/T/T
$W2$-$D1$	No	No	0.60	0.00	T/T/T	Yes	No	0.82	1.00	F/T/T
$W2$-$D2$	No	No	0.59	0.00	T/F/T	Yes	No	0.80	1.00	F/T/T
$W2$-$D3$	No	No	0.62	0.04	T/F/F	Yes	No	0.83	1.00	F/T/T
$W2$-$D4$	Yes	No	0.60	0.06	T/F/F	Yes	No	0.80	0.96	F/T/T
$W3$-$D1$	No	No	0.62	0.03	F/F/F	No	No	0.85	1.00	F/T/T
$W3$-$D2$	No	No	0.61	0.12	T/F/F	No	No	0.88	1.00	F/T/T
$W3$-$D3$	No	No	0.62	0.10	F/F/F	No	No	0.82	1.00	F/T/T

Table 4: The results for log-based approaches.

	W_{ARNE}					$W_{petrify}$				
Name	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$	**BS**	**R**	**S**	β	$\alpha/\gamma/\delta$
$W1$-$D1$	Yes	No	0.79	1.00	T/T/T	Yes	No	0.80	1.00	T/T/T
$W1$-$D2$	Yes	No	0.83	1.00	T/T/T	Yes	No	0.79	1.00	T/T/T
$W1$-$D3$	Yes	No	0.81	1.00	T/T/T	Yes	No	0.74	1.00	T/T/T
$W1$-$D4$	Yes	No	0.78	1.00	T/T/T	Yes	No	0.71	1.00	T/T/T
$W2$-$D1$	Yes	No	0.79	0.15	T/T/T	No	No	0.74	1.00	T/T/T
$W2$-$D2$	Yes	No	0.76	0.01	T/T/T	No	No	0.74	0.54	T/T/T
$W2$-$D3$	Yes	No	0.79	0.01	T/T/T	No	No	0.74	1.00	T/T/T
$W2$-$D4$	Yes	No	0.79	0.01	T/T/T	No	No	0.73	1.00	T/T/T
$W3$-$D1$	–	–	–	–	–	No	Yes	0.79	0.02	T/T/T
$W3$-$D2$	–	–	–	–	–	No	Yes	0.69	1.00	T/T/T
$W3$-$D3$	–	–	–	–	–	No	Yes	0.75	0.11	T/T/T

Table 5: The results for automata-based approaches.

8.2 Results

The results of our evaluation are reported in Tables 4 and 5, while models created by ARNE are visible in Appendix I. 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 $\alpha = \gamma = \delta = true$. On the other hand, both failing on these measures, log-based discovery and repair 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 behaviours in β.

The repair based technique instead shows more consistent results. It scores very well in **S**, β, and it only fails in α. This was expected: indeed, Repair Model ProM plug-in is based on alignment, whose focus is on adding behaviors by modifying the model (moves on the model) according to the log traces. When the rules span parallel branches, then the Repair Model ProM plug-in is not able to perform repair operations and the model stays the same (high **S** score). Then, by using the *removed unused part* option, some of the unused parts, which all belongs to α, are removed, but others are not.

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 datasets: for both techniques, we can observe that it is either extremely low (close to 0) or extremely good (close 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. Parallel branches allow for many possibilities of interleaving activities, by synchronising these branches we remove a lot of combinations. 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 property 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 procedural process models which satisfy a set of declarative business 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.

9 Summary

In this thesis, 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 emphasise the soundness of the automata-based approaches, but also bring open questions and an opportunity for future investigations. In particular, this thesis 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 models versus other approaches to combined models.

References

[1] Armas-Cervantes, A., Rosa, M. L., Menjivar, M. D., García-Bañuelos, L., and van Beest, N. R. Interactive and incremental business process model repair. April 2017.

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

[3] Baier, C., and Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] Buijs, J. C. A. M., La Rosa, M., Reijers, H. A., van Dongen, B. F., and van der Aalst, W. M. P. *Improving Business Process Models Using Observed Behavior*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 44–59.

[5] Burattin, A. PLG2: multiperspective process randomization with online and offline simulations. In *BPM Demos* (2016), pp. 1–6.

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

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

[8] De Giacomo, G., Dumas, M., Maggi, F. M., and Montali, M. *Declarative Process Modeling in BPMN*. Springer International Publishing, Cham, 2015, pp. 84–100.

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

[10] De Giacomo, G., and Vardi, M. Y. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013* (2013), pp. 854–860.

[11] De Smedt, J., De Weerdt, J., Vanthienen, J., and Poels, G. Mixed-paradigm process modeling with intertwined state spaces. *Business & IS Eng. 58*, 1 (2016), 19–29.

[12] Dijkman, R. M., Dumas, M., and Ouyang, C. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol. 50*, 12 (Nov. 2008), 1281–1294.

[13] Dijkman, R. M., Dumas, M., van Dongen, B. F., Käärik, R., and Mendling, J. Similarity of business process models: Metrics and evaluation. *Inf. Syst. 36*, 2 (2011), 498–516.

[14] Dumas, M., Rosa, M. L., Mendling, J., and Reijers, H. A. *Fundamentals of Business Process Management*. Springer Publishing Company, Incorporated, 2013.

[15] Fahland, D., and van der Aalst, W. M. Model repair - aligning process models to reality. *Inf. Syst. 47*, C (Jan. 2015), 220–243.

[16] Gambini, M., La Rosa, M., Migliorini, S., and Ter Hofstede, A. H. M. *Automated Error Correction of Business Process Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 148–165.

[17] Hildebrandt, T. T., Mukkamala, R. R., Slaats, T., and Zanitti, F. Contracts for cross-organizational workflows as timed dynamic condition response graphs. *J. Log. Algebr. Program. 82*, 5-7 (2013), 164–185.

[18] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson/Addison Wesley, Boston, 2007.

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

[20] Lohmann, N., and Fahland, D. *Where Did I Go Wrong?* Springer International Publishing, Cham, 2014, pp. 283–300.

[21] Maggi, F. M., Slaats, T., and Reijers, H. A. *The Automated Discovery of Hybrid Processes*. Springer International Publishing, Cham, 2014, pp. 392–399.

[22] Moody, D. L. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE TSE 35*, 6 (2009), 756–779.

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

[24] Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., and Reijers, H. A. *Imperative versus Declarative Process Modeling Languages: An Empirical Investigation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 383–394.

[25] Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1977), SFCS '77, IEEE Computer Society, pp. 46–57.

[26] Polyvyanyy, A., García-Bañuelos, L., Fahland, D., and Weske, M. Maximal structuring of acyclic process models. *Comput. J. 57*, 1 (2014), 12–35.

[27] Reijers, H. A., Slaats, T., and Stahl, C. *Declarative Modeling–An Academic Dream or the Future for BPM?* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 307–322.

[28] Slaats, T., Schunselaar, D. M. M., Maggi, F. M., and Reijers, H. A. *The Semantics of Hybrid Process Models*. Springer International Publishing, Cham, 2016, pp. 531–551.

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

[30] van der Aalst, W. M. P. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[31] Weijters, A., and Aalst, W. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering 10*, 2 (2003), 151–162.

[32] Westergaard, M. *Better Algorithms for Analyzing and Enacting Declarative Workflow Languages Using LTL*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 83–98.

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

[34] Westergaard, M., and Slaats, T. *Mixing Paradigms for More Comprehensible Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 283–290.

[35] Zur Muehlen, M., and Indulska, M. Modeling languages for business processes and business rules: A representational analysis. *Inf. Syst. 35*, 4 (2010), 379–390.

Appendix

I. Procedural Models

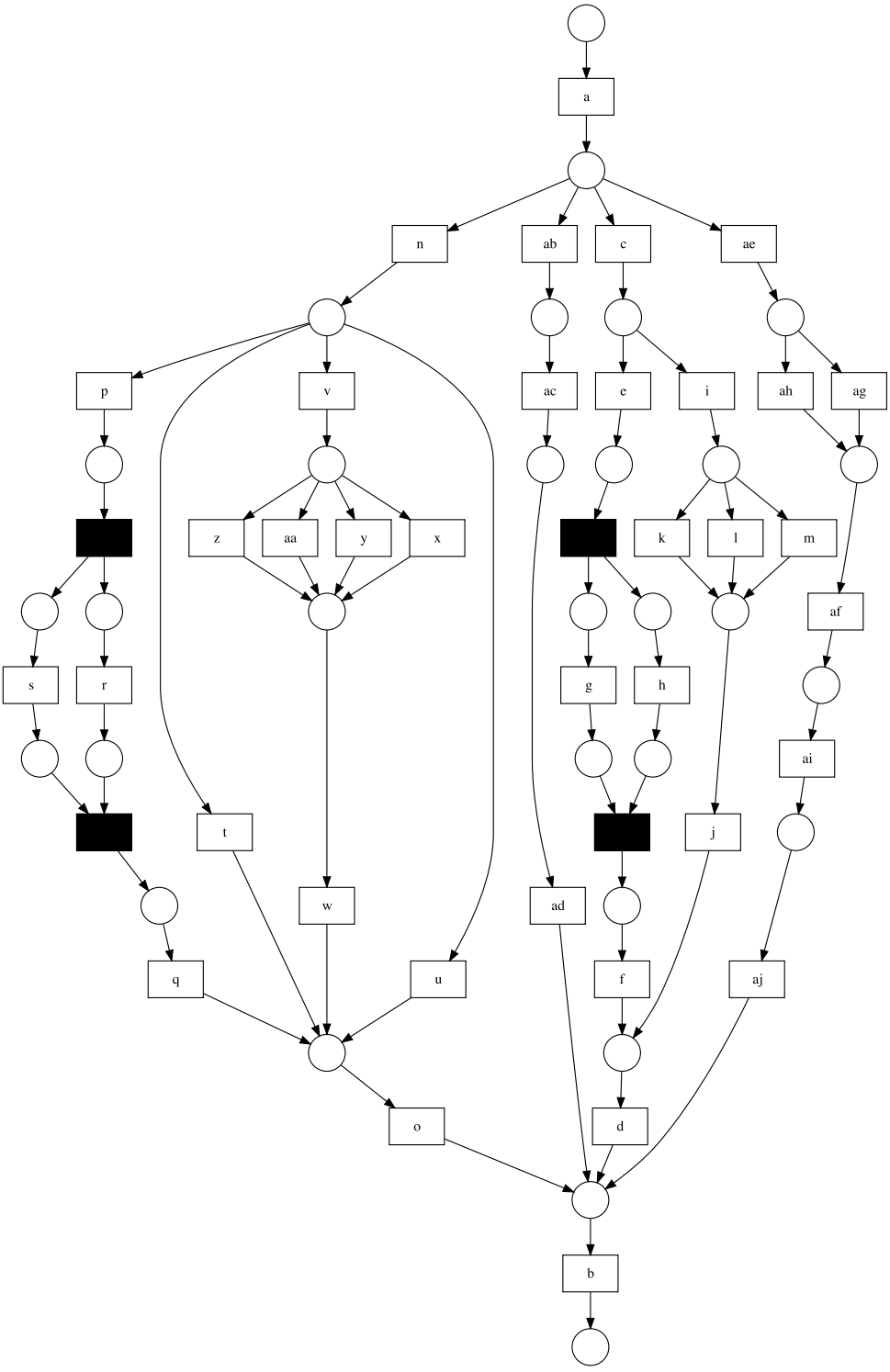

Figure 18: $W1$-XOR

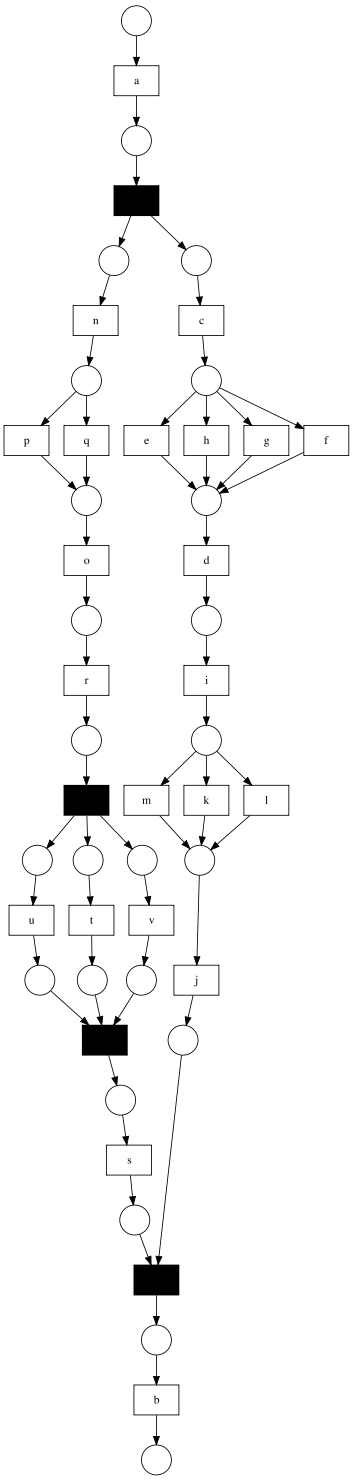

Figure 19: $W2$-AND

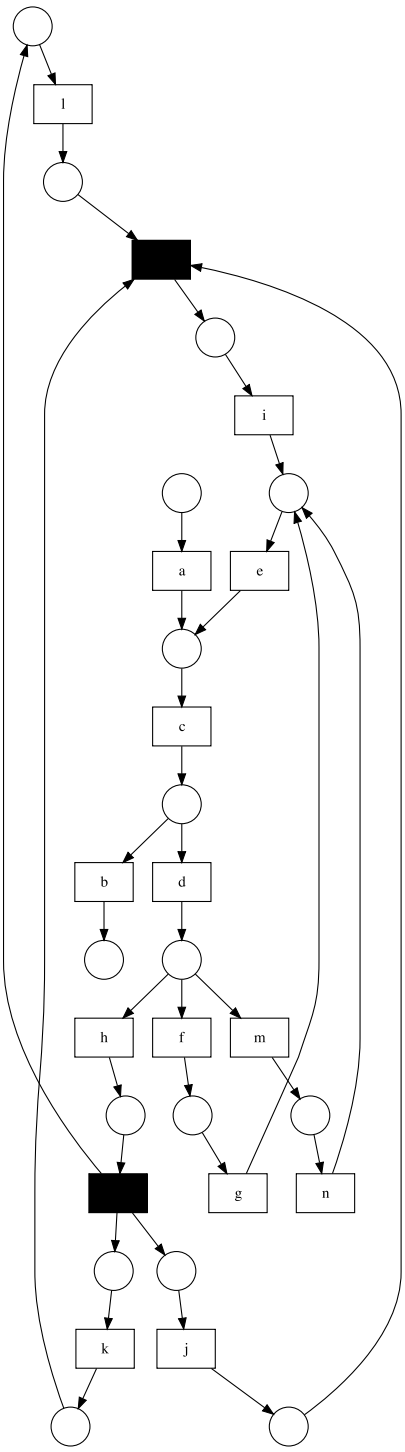

Figure 20: $W3$-LOOP

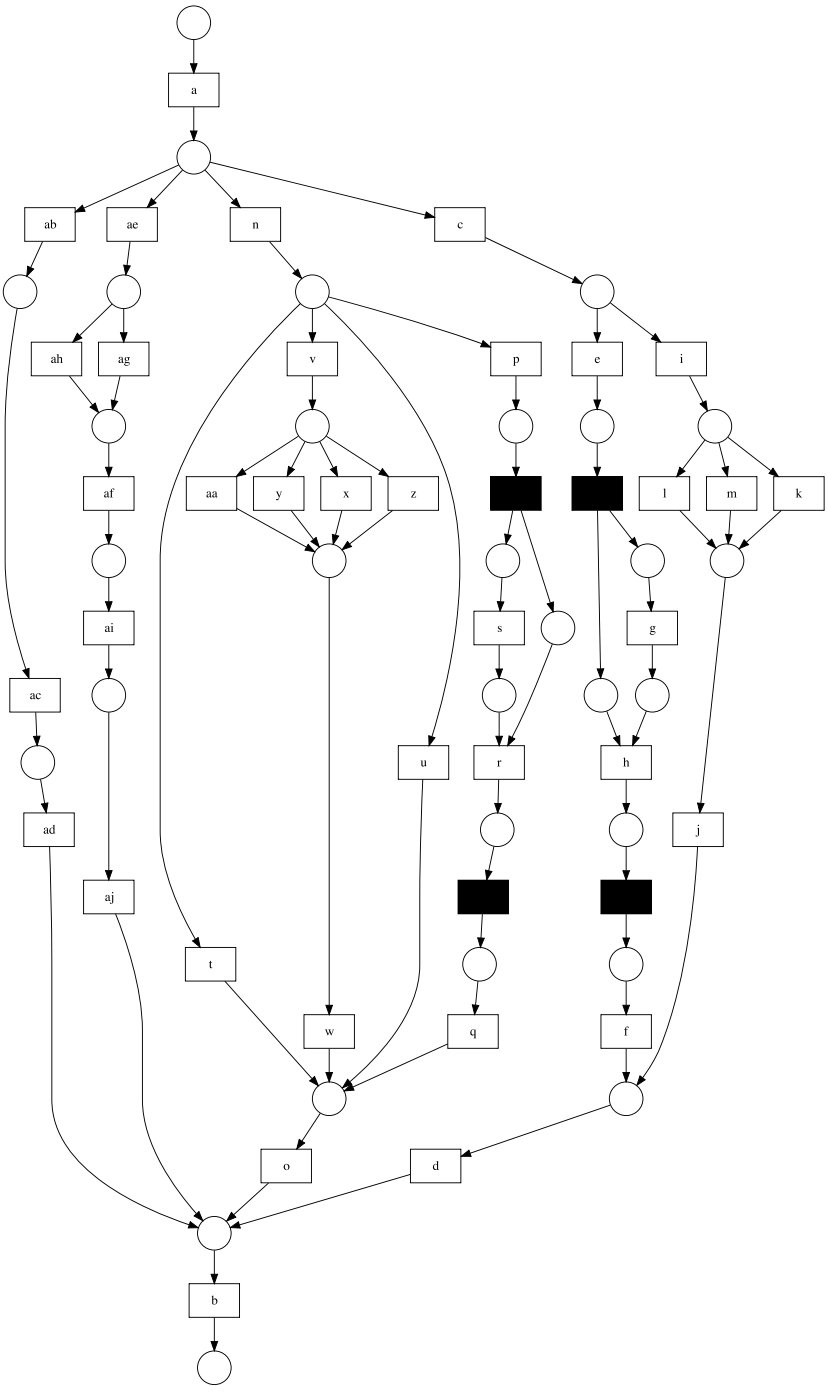

Figure 21: Resulting model of ARNE for rule $W1$-$D1$

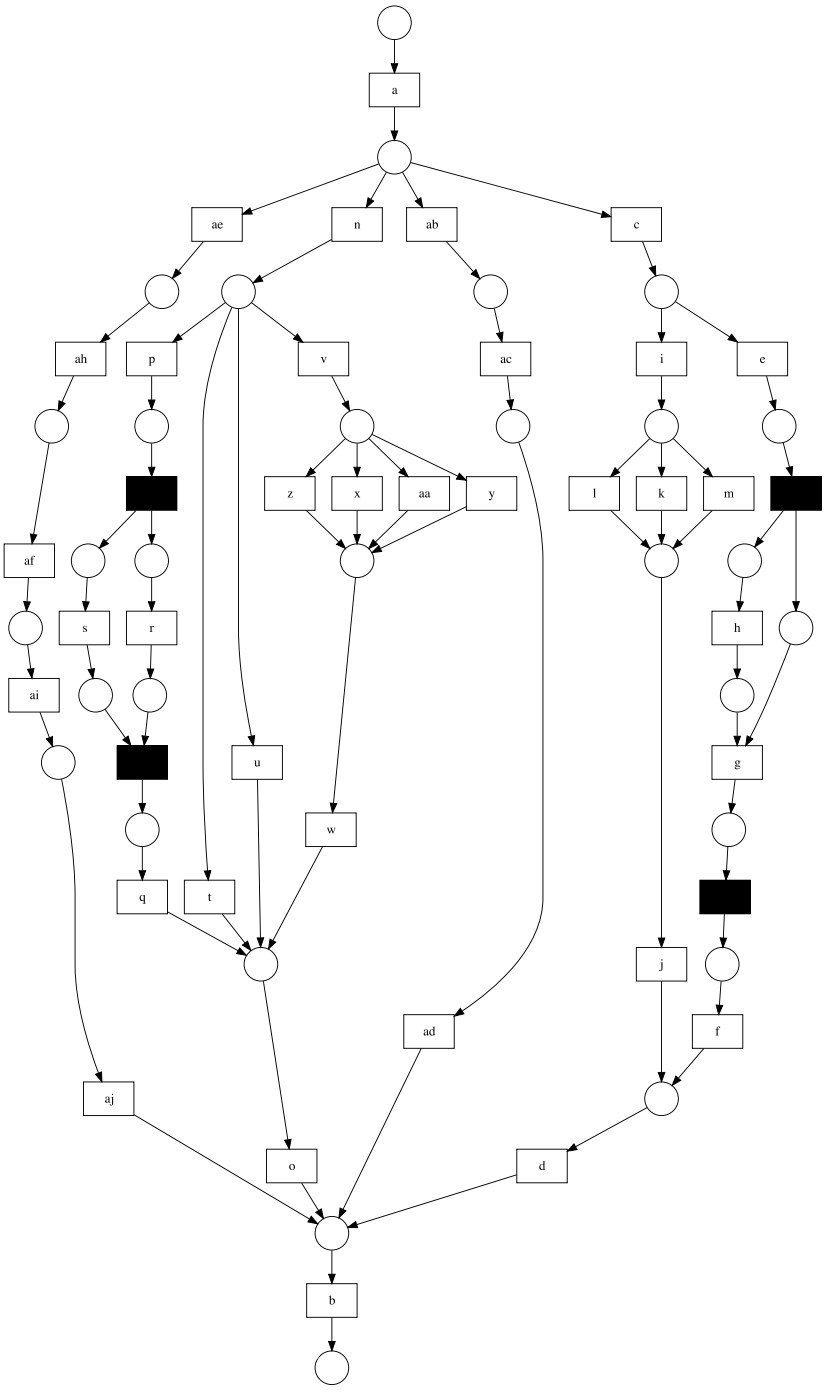

Figure 22: Resulting model of ARNE for rule $W1$-$D2$

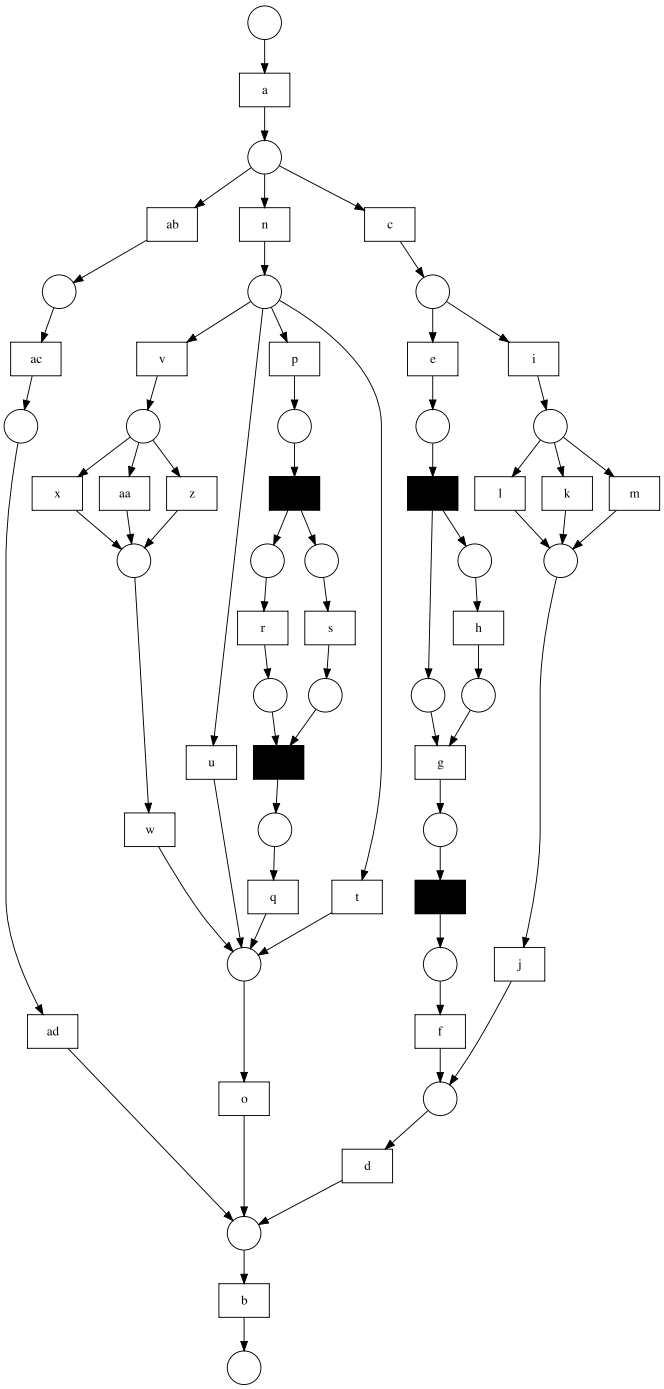

Figure 23: Resulting model of ARNE for rule $W1$-$D3$

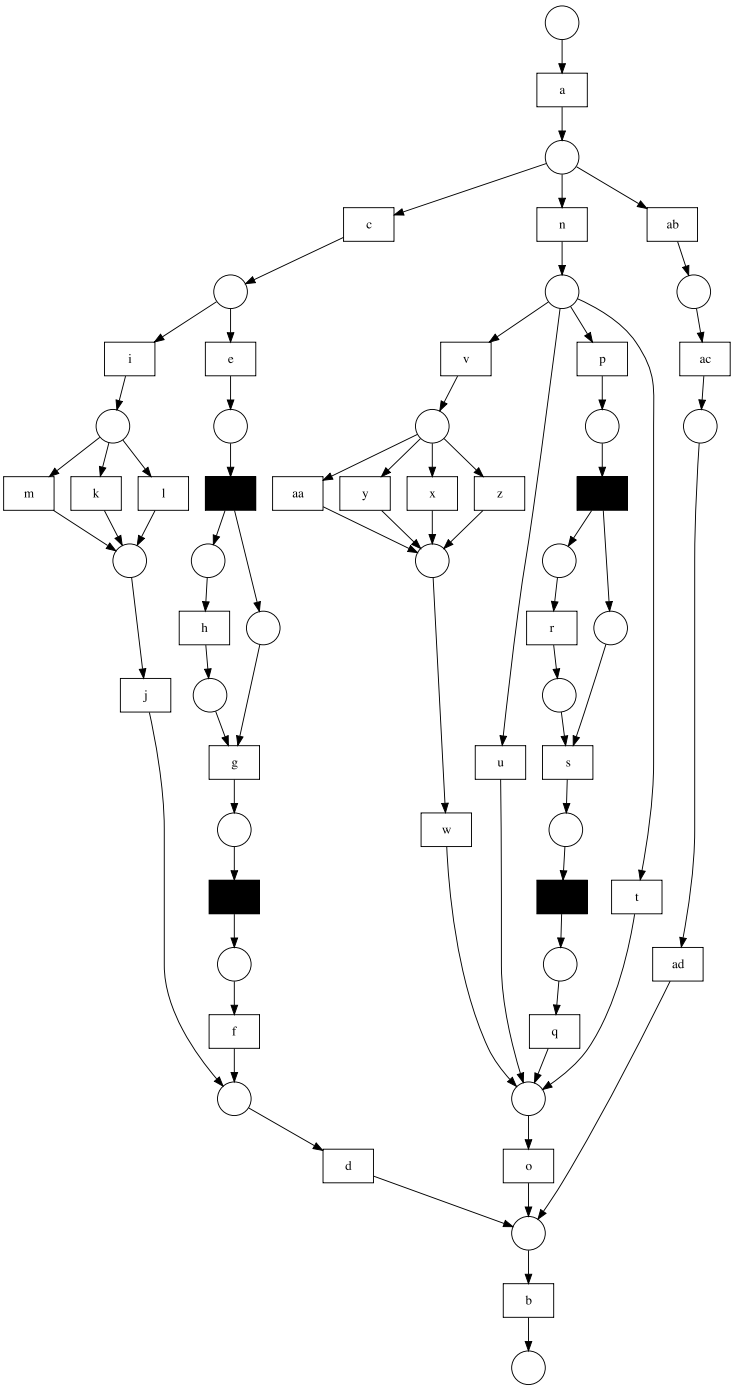

Figure 24: Resulting model of ARNE for rule $W1$-$D4$

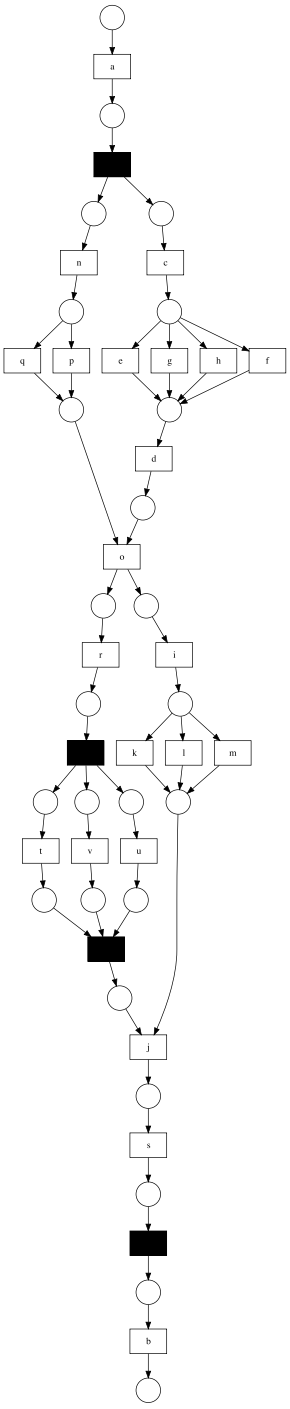

Figure 25: Resulting model of ARNE for rule $W2$-$D1$

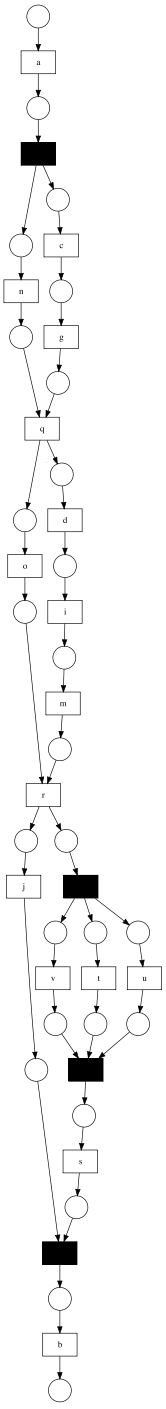

Figure 26: Resulting model of ARNE for rule $W2$-$D2$

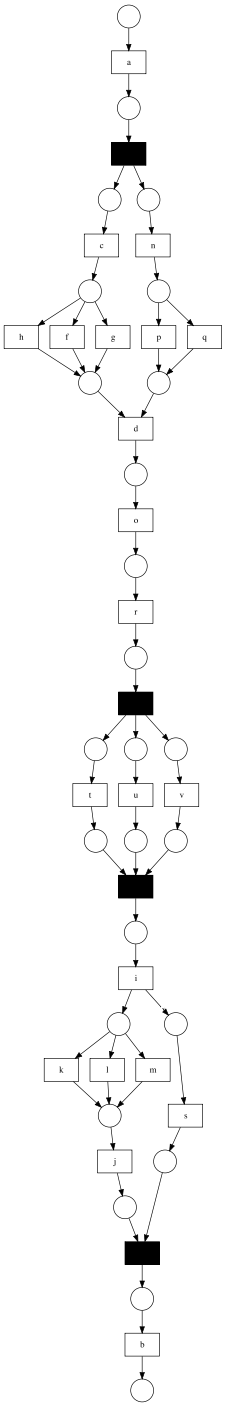

Figure 27: Resulting model of ARNE for rule $W2$-$D3$

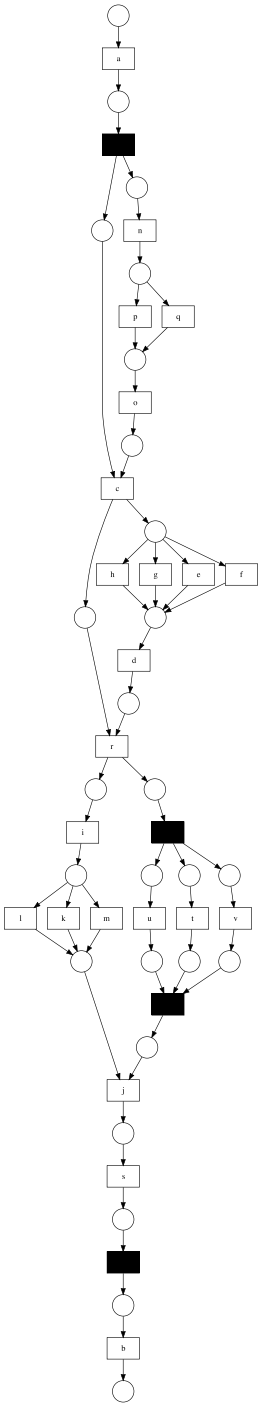

Figure 28: Resulting model of ARNE for rule $W2$-$D4$

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Arne Lapõnin**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Propagating Changes between Declarative and Procedural Process Models

   supervised by Fabrizio Maria Maggi, Riccardo De Masellis and Chiara Di Francescomarino

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 17.05.2017