Declarative vs procedural languages for data-aware compliance checking a case study in finance

Schouten, M.H.M.

Award date:
2015

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group
Eindhoven University of Technology

DECLARATIVE VS PROCEDURAL LANGUAGES FOR DATA-AWARE COMPLIANCE CHECKING: A CASE STUDY IN FINANCE

by
M.H.M. SCHOUTEN, BSC

Assessment Committee Members

dr. M. de Leoni (TU\e)
ir. M. Dees (UWV)
dr. ir. H. Eshuis (TU\e)
dr. N. Zannone (TU\e)

Eindhoven, October 2015

ABSTRACT

Process mining is the research discipline that provides techniques that can be used to discover, conformance check and enhance business process models using event logs. This thesis presents a case study done in collaboration with UWV, an autonomous administrative authority that implements employee insurances and provides labour market and data services in the Netherlands. The case study answers multiple business questions regarding the WIA application process, both using procedural and declarative process mining techniques. The WIA is an employment insurance in the Netherlands for clients that are still (partially) unfit for work after two years of illness. This thesis provides a comparative analysis of employing procedural and declarative languages that are used to perform an analysis of WIA application proces. In the WIA application process multiple decision points are present that are governed by conditions. Data-aware compliance checking techniques are required to correctly handle the decision points that are governed by conditions in the case study. Data-aware compliance checking techniques are currently non existent for declarative process models. This thesis presents a new data-aware compliance checking technique for declarative process models. The presented technique provides analysts with the possibility to determine how compliant a declarative process model is with a given event log by presenting the fulfillment ratio (i.e. the ratio of traces not violating the constraint) of every constraint. The thesis validates the technique by using synthetic created event logs. Assessing the technique using the real-life event log extracted for the UWV case study shows that the technique has a practical feasibility and relevance and that it is able to produce similar results to existing procedural techniques.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisior Massimiliano de Leoni for the feedback, advice and guidance during my master thesis. Your help has been valuable and led to the work I present in this thesis. Additionally I would like to thank Nicola Zannone and Rik Eshuis for joining the assessment committee.

I would like to thank everyone at UWV for the nice working atmosphere during the project. Especially I would like to thank Henk de Ruiter and Marcus Dees for giving me the possibility of doing a case study at UWV. The possibility of working on a project in a workplace with real-life data, plus presenting results to the board of directors has been a valuable experience I would not have liked to have missed. Additionally I would like to thank Marcus for his extensive help during the case study, your SQL and process mining knowledge have been a great help in working on a real-life case study. I would also like to thank Loes Bilderbeek and Marije Dijkstra for the numerous interviews, the feedback and the provision of information regarding the WIA process.

I thank all of my friends, roommates and colleagues for the great time in Eindhoven. I would like to thank my parents and sister Jamie for their support throughout the years. Last but not least, I would like to thank my girlfriend Anne for her support the last years.

CONTENTS

LIST OF TABLES

ACRONYMS

UWV Employee Insurance Agency

SZW Ministry of Social Affairs and Employment

WW Unemployment Insurance Act

WIA Work and Income according to Labour Capacity Act

LTL Linear Temporal Logic

1 INTRODUCTION

Over the years companies have been using information systems more and more to support and control business processes. Information systems allow for recording and storing data related to business processes in the form of *event logs*. The goal of *process mining* is to extract value from these event logs by obtaining process related information [30].

An event log is a collection of sequentially recorded events such that each event refers to an *activity* (i.e., a well defined step in the process) and is related to a particular *case* (i.e., a process instance). A sequence of events belonging to a single case is called a *trace*. Most event logs store additional information about events, such as the *resource* (i.e., person or device) executing the activity, the *timestamp* , or *data attributes* recorded with the event (e.g. the office in which the task is executed) [30].

Process mining can usually be distinguished in three different types: *process discovery*, *conformance checking* and *enhancement*. A *process discovery* technique takes an event log and produces a model explaining behavior recorded in the log. When using *conformance checking* an existing process model is compared with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa. The idea of *enhancement* is to extend or improve an existing process model using information about the actual process as recorded in the log [30].

One of the open challenges in process mining is to find a suitable representational bias (language) to visualize the resulting models [24]. The suitability of a language largely depends on the level of standardization and the environment of the process. Standardized processes in stable environments (e.g., a process for handling insurance claims) are characterized by low complexity of collaboration, coordination and decision making. In addition, they are highly predictable, meaning that it is feasible to determine the path that the process will follow. On the other hand, processes in dynamic environments are more complex and less predictable. They comprise a very large number of possible paths as process participants have considerable freedom in determining the next steps in the process (e.g., a doctor in a healthcare process)[30].

As discussed in [32], [25], procedural languages, such as BPMN, EPCs and Petri nets, are suitable for describing standardized processes in stable environments. In contrast, the use of procedural languages for describing processes in dynamic environments leads to complex and incomprehensible models. In this context, declarative process modeling languages are more appropriate [32]. Over the last years, the declarative process modeling approach has flanked the classical procedural one [32], [25]. Declarative approaches only depict the behavioural constraints under which a process instance can unfold in its execution: as long as the constraints are not violated, the process instance is considered as valid. The declarative approach is a complementary strategy to the procedural models, which specify what are the next allowed activities at each stage of the process execution. Declarative process models are effective in a context of high flexibility for business processes [27]. The reason intuitively lies in the fact that fewer constraints allow for more possible executions. On the contrary, more flexibility implies a higher number of alternative paths to depict in the procedural models.

Declare [22] is a declarative process modeling language. Declare is a declarative language that combines a formal semantics grounded in Linear Temporal Logic (LTL) on finite traces, with a graphical representation. In essence, a Declare model is a collection of LTL rules, each capturing a control-flow dependency between two activities [11].

In this thesis all distinguished three types of process mining will be touched, but particularly process discovery and conformance checking is addressed. More specifically, this thesis provides a comparative analysis of employing procedural and declarative languages that are used to perform an analysis of some business processes enacted at Employee Insurance Agency (UWV).

UWV is an autonomous administrative authority and is commissioned by the Ministry of Social Affairs and Employment (SZW) to implement employee insurances and provide labour market and data services in the Netherlands. More specifically, UWV has core tasks in the following four areas:

- Employment - helping the client remain employed or find employment, in close cooperation with the municipalities;
- Social medical affairs - evaluating illness and labour incapacity according to clear criteria;
- Benefits - ensuring that benefits are provided quickly and correctly if work is not possible, or not immediately possible;
- Data management - ensuring that the client needs to provide the government with data on employment and benefits only once.

The vision of UWV is that people are at their best when they can participate in society by working. Society functions best when as many people as possible participate in it by working. It is the mission of UWV to make a difference for people by promoting work and in case work is impossible, UWV ensures that income is available quickly.
The Dutch employee insurances are provided for via laws such as the Unemployment Insurance Act (WW) and the Work and Income according to Labour Capacity Act (WIA). The case study of this thesis will focus on the WIA Claim process in particular.

The WIA is an employment insurance in the Netherlands for clients that are still (partially) unfit for work after two years of illness. During the first two years of illness the employer of the client is obliged to pay the wages of the client. After these two years of illness a client can apply for benefits, after which UWV will assess the application to determine whether it will be accepted. This assessment process is called the WIA Claim process. The term client is adapted from UWV as this is the common term used in the company.

An extracted event log at UWV is used to answer business questions for UWV and is used to evaluate the developed techniques in this thesis.

1.1 SCOPE

Recently UWV has started adapting to process mining as a method of analysing the business processes. For this reason it was chosen to use process mining techniques to answer several business questions regarding the WIA Claim process. To answer some of the business questions conformance checking is used. In the WIA Claim process multiple decision points are present that are governed by conditions. For this reason multiple data attributes are taken into account when answering business questions using conformance checking.

For this case study first and foremost a procedural process model of the WIA Claim process is drawn, which is used to answer the business questions. To answer the business questions using a procedural model, multiple process mining techniques are available to add guards to procedural models and to do conformance checking while taking guards into account [9] [20] [16]. Guards are conditions that state under what conditions a task should or should not be executed. Beside the procedural approach, UWV is interested in whether a declarative approach is better suited to model their business processes. For this reason a declarative process model of the WIA Claim

process is created. For declarative models a conformance checking approach is available that focuses solely on the control-flow [19]. However, focussing solely on the control-flow results in the technique being unable to determine whether deviations are present concerning data attributes.

Since for some business processes a declarative approach might be more appropriate [32], a technique that is able to diagnose deviations concerning data attributes is desired for declarative approaches. Solely using control-flow conformance checking without considering data attributes results in unjust violations. This issue leads to the following problem statement that is addressed in this thesis:

PROBLEM STATEMENT Data-aware compliance checking is essential in analyzing business processes which contain decision points that are governed by conditions. Declarative process models are effective in a context of high flexibility for business processes, but data-aware compliance checking techniques are currently non existent for declarative process models. Data-aware compliance checking of declarative process models is desired since not considering data attributes for processes that contain decision points that are governed by conditions results in unjust violations.

This thesis will present a technique which is able to address the problem statement for declarative process models in the Declare language. This technique is motivated by the following research goal:

RESEARCH GOAL Develop a technique that allows for data-aware compliance checking of Declare models with guards, such that data-aware analysis can be done for declarative approaches similar to procedural approaches.

The research goal is achieved by delivering a technique that allows for data-aware compliance checking for Declare models given an event log and a Declare model enriched with guards. The thesis limits to a compliance checking technique that determines for each constraint in the Declare model the set of traces that violate the constraint, i.e. the technique does not determine optimal alignments for each trace like a conformance checking technique does. In the case study of this thesis only the detection of violations is required, i.e. no optimal alignments are required. With this technique, similar to a data-aware procedural approach, data-aware compliance checking can be done for a declarative approach. Hence, allowing a declarative approach which is more suitable to flexible business processes to diagnose deviations concerning data attributes.

To assess the practical feasibility and relevance, the technique has been implemented in ProM. ProM is an extensible framework that provides support to develop and exploit a wide variety of process mining techniques in a standardised environment [12]. The implementation of the technique has been validated and evaluated using synthetic event logs and the real life event log extracted at UWV. The output presented by this technique has been integrated in the correlation framework presented in [21]. Through this integration, it is now possible to correlate the violations of Declare constraints with other process characteristics, such as the occurrence of undesired events, the execution time or, even, the violations of other business rules.

Relevant to the case study, an existing implementation of a Declare miner named MINERful [7] has been implemented in ProM. The motivation for this implementation is that existing Declare miners in ProM could not scale with the size of the event logs extracted at UWV. This implementation also led to a publication accepted for the Demo Track of the BPM 2015 conference [5].

Thanks to the new contributions of this thesis some of the business questions can now be answered more accurately using a declarative approach. Experimental results show that the newly presented technique is able to produce results that are similar to the results produced by procedural techniques for the case study.

1.2 OUTLINE

Before the case study and data-aware compliance checking contributions are discussed, Chapter 2 addresses preliminary work and concepts that are relevant to the topics addressed in this thesis.

Chapter 3 addresses the case study done for UWV. The WIA Claim process is explained, as well as the design decisions that are made in extracting the event log used in the case study. In multiple interviews with WIA Claim domain experts, four business questions that need to be answered are defined. In this chapter the four business questions are answered using a procedural approach.

Chapter 4 presents the implementation of a Declare miner utilizing the MINERful algorithm in ProM. The chapter gives a short overview of the MINERful algorithm and its advantages over other declarative process discovery techniques. Usage of the tool is demonstrated using a real-life event log.

Chapter 5 presents the new contributions that allow for data-aware compliance checking of Declare models. The chapter presents an algorithm that is used to determine whether a specific trace violates a Declare constraint. The usage of the two new contributions is show

cased on an example model with an example log. Finally the chapter validates the contributions using multiple test event logs.

Chapter 6 addresses the declarative part of the case study done for UWV. The declarative part of the case study at the same time serves as an evaluation of the new contributions, by assessing the practical feasibility and relevance using a real-life event log. In this chapter three of the four business questions are answered using the new declarative techniques.

Finally the thesis is concluded in Chapter 7. Chapter 7 addresses the conclusions, presents an advice for UWV based on the case study and addresses possible future work.

2

STATE OF THE ART

In process mining, models are used for explaining behavior recorded in an event log or are used to specify behavior that is allowed in a system or process. Over the years multiple modelling languages have been created, all of which have their own strengths and weaknesses. Models can be useful to understand, define, visualize or simulate processes. As models might contain oversimplifications or illogical assumptions, models cannot be used as an exact copy of reality. Nevertheless models are suited to be used as a reflection, as opposed to a copy, of reality. In this thesis three modelling languages are used, namely Declare, final-state automata and Petri nets. In this section state of the art knowledge of the mentioned modelling languages Declare and Petri nets as well as multiple process mining techniques are explained. The contents of this section are a preliminary introduction as the content is used and/or extended in the remainder of this thesis.

2.1 PETRI NETS

A Petri net is a mathematical modelling language that can be used to describe distributed systems. Petri nets have a strong mathematical basis, which allows for a precise analysis of a modeled system or process. Additionally Petri nets are represented graphically, making them accessible for analysts who are no modeling experts [29].

DEFINITION 1 (PETRI NET) *A Petri net is a triplet* (P, T, F) *with:* P = *a finite set of places,* T = *a finite set of transitions where* $(P \cap T = \emptyset)$ *and,* F = *the flow relation where* $F \subseteq (P \times T) \cup (T \times P)$

A place p is an input place of a transition t iff $(p, t) \in F$ and a place p is an output place of t iff $(t, p) \in F$. A marking M of a Petri net is a multiset of tokens, i.e., a mapping $M : P \to \mathbb{N}$. A marking M assigns a number of tokens to each place.

Firing a transition t in a marking M consumes one token from each of its input places and produces one token in each of its output places. A transition t is allowed to fire, i.e. is enabled, in M if there is at least one token in all of its input places to consume, i.e. for each input place S of t holds $M(S) \geqslant 1$ [13].

To correspond transitions to an activity in a process, each of these transitions are associated with a label that indicates the activity it rep-

resents. Transitions with no label are known as τ-transitions, which are invisible transitions. τ-transitions are introduced for routing purposes and do not represent an actual activity of the process. As such, τ-transition executions are not recorded in the event logs [20].

DEFINITION 2 (LABELED PETRI NET) *A labeled Petri net* $PN = (P, T, F, l)$ *is a Petri net* (P, T, F) *with labeling function* $l \in T \nrightarrow \mathcal{U}_A$ *where* $\mathcal{U}_A$ *is some universe of activity labels*

DEFINITION 3 (SYSTEM NET) *A system net* $SN = (PN, M_{init}, M_{final})$ *is a triplet where* $PN = (P, T, F, l)$ *is a labeled Petri net,* $M_{init} \in P \to \mathbb{N}$ *is the initial marking, and* $M_{final} \in P \to \mathbb{N}$ *is the final marking.* $\mathcal{U}_{SN}$ *is the universe of system nets* [20].

2.1.1 *Data Petri nets*

A Petri net with data (DPN-net) is a Petri net in which transitions can read and/or write variables [15]. A transition t can fire only if its guard is satisfied and all input places are marked, i.e. there is at least one token in all of its input places to consume. A guard can be any formula over the process variables using relational operators $(<, >, =)$ as well as logical operators such as conjunction $(\wedge)$, disjunction $(\vee)$, and negation $(\neg)$ [9].

DEFINITION 4 (VARIABLES AND VALUES) $\mathcal{U}_{VN}$ *is the universe of variable names.* $\mathcal{U}_{VV}$ *is the universe of values.* $\mathcal{U}_{VM} = \mathcal{U}_{VN} \nrightarrow \mathcal{U}_{VV}$ *is the universe of variable mappings.*

A DPN-net is formally defined as follows:

DEFINITION 5 (DPN-NET) *A Petri net with data*
$DPN = (SN, V, val, init, read, write, guard)$ *consists of:*

- *a system net* $SN = (PN, M_{init}, M_{final})$ *with* $PN = (P, T, F, l)$,
- *a set* $V \subseteq \mathcal{U}_{VN}$ *of data variables,*
- *a function* $val \in V \to \mathcal{P}(\mathcal{U}_{VV})$ *that defines the values admissible for each variable* $v \in V$*, i.e.* $val(v)$ *is the set of values that variable* v *can have,*
- *a function* $init \in V \to \mathcal{U}_{VV}$ *that defines the initial value for each variable* v *such that* $init(v) \in val(v)$ *(initial values are admissible),*
- *a read function* $read \in T \to \mathcal{P}(V)$ *that labels each transition with the set of variables that it reads,*

- *a write function* $write \in T \to \mathcal{P}(V)$ *that labels each transition with the set of variables that it writes,*
- *a guard function* $guard \in T \to Formulas(V_W \cup V_R)$ *that associates a guard with each transition such that, for any* $t \in T$ *and for any* $v \in V$*, if* v *appears in* $guard(t)$ *then* $v \in read(t)$ *and if* v' *appears in* $guard(t)$ *then* $v \in write(t)$*.*

$\mathcal{U}_{DPN}$ *is the universe of Petri nets with data [20].*

When a variable $v \in V$ appears in a guard $guard(t)$, it refers to the value just before the occurrence of t. If $v \in write(t)$, it can also appear as v' (i.e., with the prime symbol). In this case, it refers to the value after the occurrence of t [9].
A binding is a triplet (t, r, w) which describes the execution of transition t while reading values r and writing values w. A binding is valid if:

1. $r \in read(t) \to \mathcal{U}_{VV}$ and $w \in write(t) \to \mathcal{U}_{VV}$,
2. for any $v \in read(t) : r(v) \in val(v)$, i.e. all values read should be admissible,
3. for any $v \in write(t) : w(v) \in val(v)$, i.e. all values written should be admissible,
4. Guard $guard(t)$ evaluate true [20].

To illustrate, consider the following (simplified) process to request loans which is taken from [9]; The process starts with a credit request where the requester provides some documents to demonstrate the capability of paying the loan back. These documents are verified and the interest amount is also computed. If the verification step is negative, a negative decision is made, the requester is informed and, finally, the negative outcome of the request is stored in the system. If verification is positive, an assessment is made to take a final decision. Independently of the assessment's decision, the requester is informed. Moreover, even if the verification is negative, the requester can renegotiate the loan (e.g. to have lower interests) by providing further documents or by asking for a smaller amount. In this case, the verification-assessment part is repeated. If both the decision and verification are positive and the requester is not willing to renegotiate, the credit is opened.

Figure 1, which is also taken from [9], shows the DPN-net that models the loan request process. Table 1 lists the conditions of the guards of the transitions. The labeling function l is such that the domain of l is the set of transitions of the DPN-net. For each transition t of the

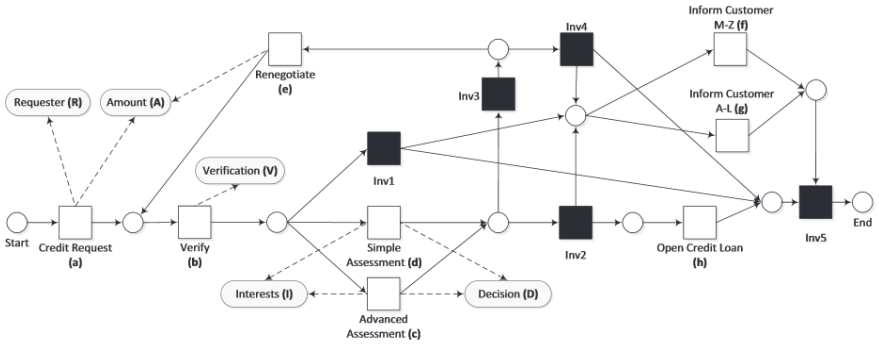

Figure 1: Pictorial representation of a Petri net with Data that models the process to request loans. Places, transitions and variables are represented as circles, rectangles and rounded rectangles, respectively. The dotted arcs going from a transition to a variable denote the writing operations; the reverse arcs denote the read operations, i.e. the transition requires accessing the current variables' value [20].

DPN-net, $l(t) = t$ stating that the set of activity labels coincides with the set of transitions.

2.2 CONFORMANCE CHECKING FOR PROCEDURAL MODELS

Often process models, such as Petri nets, are not enforced and deviated from in practice. Comparing an existing process model with an event log of the same process is conformance checking. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa [30].

Over the years multiple conformance checking techniques have been proposed [30] [2] [1]. These conformance checking techniques focus on the control-flow, i.e. the order in which activities occur. Additionally these conformance checking techniques focus on *fitness*. A model with good fitness allows for the behavior seen in the event log. A model has a perfect fitness if all events of all traces in the log can be replayed. Fitness can be defined in multiple ways, e.g. on case level (fraction of traces in the log that can be fully replayed) or event level (the fraction of events in the log that are possible according to the model) [30].

Given a process model $\mathcal{P}$ and an event log $\mathcal{L}$, deviations in the fitness are either skipped activities or inserted activities. Skipped activities refer to activities that should be performed according to the model, but do not occur in the log. Inserted activities refer to activities that occur in the log, but do not occur in the model [1]. An *alignment* shows how the event log can be replayed on the process model.

TRANSITION	GUARD
Advanced Assessment	Verification = true $\wedge$ Amount > 5000 $\wedge$ 0.1 < Interest' / Amount < 0.15
Inv1	Verification = false
Inv2	Decision = true
Inv3	Decision = false
Open Credit Loan	Verification = true $\wedge$ Decision = true
Inform Customer M-Z	Requester $\geqslant$ "M"
Inform Customer A-L	Requester $\leqslant$ "L"
Renegotiate	Amount' $\leqslant$ Amount
Simple Assessment	Verification = true $\wedge$ Amount $\leqslant$ 5000 $\wedge$ 0.15 < Interest' / Amount < 0.2

Table 1: Table listing the guards of the transitions in Figure 1

Let S_N be the set of (valid and invalid) firing of transitions of a labeled Petri net N with S_N. It is required to relate "moves" in the log to "moves" in the model in order to establish an alignment between a process model and an event log. However, it may be the case that some of the moves in the log cannot be mimicked by the model and vice versa. "No move" is denoted by $\gg$. For convenience, the set $S_N^{\perp} = S_N \cup \{\gg\}$ is introduced [9].

One move in an alignment is represented by a pair $(s', s'') \in (S_N^{\perp} \times S_N^{\perp}) \setminus \{(\gg, \gg)\}$ such that:

- (s', s'') is a move in log if $s' \in S_N$ and $s'' = \gg$
- (s', s'') is a move in model if $s' = \gg$ and $s'' \in S_N$
- (s', s'') is a move in both if $s' \in S_N$ and $s'' \in S_N$

Let σ_L be a trace in event log $\mathcal{L}$ and let σ_P be a full execution sequence of model $\mathcal{P}$. An alignment of σ_L and σ_P is a sequence γ such that the projection on the first element (ignoring $\gg$) yields σ_L and the projection on the second element (again ignoring $\gg$) yields σ_P. Two examples of alignments can be seen in Figure 2.

In practice, the severity of skipping or inserting activities may depend on the activity. To introduce this notion in alignment, a cost

$$\gamma_1 = \begin{array}{|c|c|c|c|c|} \hline a & c & d & e & h \\ \hline a & c & d & e & h \\ \hline \end{array} \qquad \gamma_2 = \begin{array}{|c|c|c|c|c|c|c|} \hline a & b & \gg & d & e & g & \gg \\ \hline a & \gg & c & d & e & \gg & h \\ \hline \end{array}$$

Figure 2: Alignment γ_1 is an alignment of σ_L = *acdeh* and σ_P = *acdeh*. Alignment γ_2 is an alignment of σ_L = *abdeg* and σ_P = *acdeh*. Moves are represented vertically, e.g., the first move of γ_1 is *(a,a)* indicating that both the log and the model make an *a* move.

function is introduced. The cost function can be generalized to alignments as the sum of the cost of each individual move. The goal of conformance checking is to find the alignment of log trace $\sigma \in \mathcal{L}$ and $\mathcal{P}$ that minimizes the cost. Such an alignment is called an *optimal alignment*. Creating an optimal alignment with respect to a custom cost function can be done using the A$*$ algorithm, as illustrated in [9].

2.2.1 *Data-aware Conformance Checking for Procedural Models*

Conformance checking techniques that only consider the control-flow perspective cannot find any conformance violations for data-dependent guards in DPN-nets. In recent years some data-aware conformance checking techniques have been proposed [16] [9].

When data-aware conformance checking, it can occur that alternative explanations exist for a deviating trace. For the identified deviation of some activity the explanation may be (1) that all data values were written correctly but, the activity did not need to be performed, or (2) the activity was performed properly but the data variables were not set correctly.

The approach in [16] seeks for explanations that put the control-flow first. This approach would prefer explanation (2) even when the other perspectives strongly suggest an alternative explanation with more control-flow deviations. Such explanations can be constructed quickly, at the potential expense of the inability to guarantee the optimality of the solution. Indeed, explanation (2) requires one to accept that observed data values are incorrect, which in some particular case may actually be less likely than only one activity being executed incorrectly. Hence, explanation (1) may be more likely in certain settings. This shows that there are tradeoffs between the different perspectives. The approach in [9] allows for balancing the control-flow, data, resources, and time perspectives in identifying explanations for deviations. For this reason the approach in [9] is used in this thesis.

For data-aware conformance checking, the notion of alignments is extended to take data values into account. Alignments have been defined in [9] as follows:

DEFINITION 6 (ALIGNMENTS) *Let* $DPN = (SN, V, val, init, read, write, guard)$ *be a DPN-net. A legal move in an alignment is represented by a pair* $(s', s'') \in (S_N^{\perp} \times S_N^{\perp}) \setminus \{(\gg, \gg)\}$ *such that:*

- (s', s'') *is a move in log if* $s' \in S_N$ *and* $s'' = \gg$
- (s', s'') *is a move in model if* $s' = \gg$ *and* $s'' \in S_N$
- (s', s'') *is a move in both with correct write operations if* $s' \in S_N$, $s'' \in S_N$ *and* $\#_{act}(s') = \#_{act}(s'')$ *and* $\forall v \in V \; \#_{vars}(s', v) = \#_{vars}(s'', v)$
- (s', s'') *is a move in both with incorrect write operations if* $s' \in S_N$, $s'' \in S_N$ *and* $\#_{act}(s') = \#_{act}(s'')$ *and* $\exists v \in V \; \#_{vars}(s', v) \neq \#_{vars}(s'', v)$

To explain the different perspectives relevant for conformance, consider the following example trace given in [9]: $\sigma_{example}$ = $\langle$(a, {A = 3000, R = Michael, E_a = Pete, T_a = 3Jan}), (b, {V = false, E_b = Sue, T_b = 4 Jan}), (c, {I = 530, D = true, E_c = Sue, T_c = 5 Jan}), (f, {E_f = Pete, T_f = 17 Jan})$\rangle$. Trace $\sigma_{example}$ consists of 4 events. Lower-case bold letters refer to activities using the mapping in Figure 1, e.g., a = Credit Request. Upper-case bold letters refer to data objects. A = 3000 describes that the amount is 3000 (A is a shorthand for Amount) and R = Michael describes that credit request is initiated by Michael (R is a shorthand for Requester). E_x and T_x respectively denote the last executor of x and the timestamp when x was executed last. Two example alignments for this example log $\sigma_{example}$ and the DPN-net in Figure 1 can be seen in Figure 3.

γ_1 =

Log Trace	Process
a {A = 3000, R = Michael}	a {A = 5001, R = Michael}
b {V = *false*}	b {V = *true*}
c {I = 530, D = *true*}	c {I = 530, D = *false*}
$\gg$	Inv3
$\gg$	Inv4
f {}	f {}
$\gg$	Inv5

γ_2 =

Log Trace	Process
a {A = 3000, R = Michael}	a {A = 3000, R = Michael}
b {V = *false*}	b {V = *false*}
$\gg$	Inv1
c {I = 530, D = *true*}	$\gg$
f {}	f {}
$\gg$	Inv5

Figure 3: Examples of alignments of $\sigma_{example}$ and the DPN-net in Figure 1

The cost function assigns a non-negative cost to each legal move specified in Definition 6. The cost function can be used to favor one type of explanation for deviations over the other. Beside finding an optimal alignment, an alignment that minimizes the cost, the fitness level of traces and logs is quantified. The fitness function is defined as follows:

DEFINITION 7 (FITNESS LEVEL) *Let* σ *be a log trace and* N *be a DPN-net with cost function* $\mathcal{K}$*. Let* γ_O *be an optimal alignment of* σ *and* N *and let* γ_E *be an optimal alignment of the empty trace and* N*. Let* γ_R *be the reference alignment given by* $\gamma_R = \gamma_E \oplus \langle (s_1, \gg), ..., (s_n, \gg) \rangle$ *with* $s_i \subseteq \sigma$*. The fitness level of* σ *and* N *is defined as follows:*
$\mathcal{F}(\sigma, N) = 1 - \frac{\mathcal{K}(\gamma_O)}{\mathcal{K}(\gamma_R)}$

Creating an optimal multi-perspective alignment with respect to a custom cost function that is balanced is illustrated in [9]. The technique is implemented as a plug-in in ProM named "Conformance Checking of DPN (Balanced)" and is used in the case study of this thesis.

2.2.2 *Performance Analysis for Procedural Models*

After aligning event log and model, all kinds of analysis techniques based on "replay" are possible. These replay techniques may use additional attributes and are not restricted to activity names. For the case study in this thesis time related performance analysis is required. For this reason the performance analysis technique presented in [31] is used. As mentioned in [31], timestamps of events can be used to compute flow times, waiting times, service times, synchronization times, etc. For example, let *e1* and *e2* be two subsequent events with *act(e1) = a, act(e2) = b, time(e1) = 23-11-2011:15.56*, and *time(e2) = 23-11-2011:16.20*. If b is causally dependent on a, a time of 24 minutes is recorded in between a and b. By repeatedly measuring such time differences during replay, the average time that elapses in-between a and b is computed. Such detailed analysis is only possible after successfully aligning model and log. A process model annotated with time information can be used to diagnose performance problems [31]. The time related performance analysis has been implemented as a plug-in in ProM named "Replay a Log on Petri Net for Performance/Conformance analysis (bottlenecks places or tasks)" and is used in the case study of this thesis. The plug-in is able to distinguish between four groups of performance information, namely: Process metrics, Place metrics, Two-transition metrics, and Activity metrics [26].

Process metrics

The following process-related metrics are derived by this plug-in:

- Total number selected: the total number of process instances.

- Number fitting: the number of process instances that complete properly and successfully, i.e. the number of instances that can be replayed in the Petri net without any problems.
- Arrival rate: the number of arrivals of process instances per time unit.
- Throughput time: the throughput time of the process instances.

Place metrics

The place-related metrics that are derived consist of:

- Frequency: the number of visits of tokens to the place during replay of the process instances in the Petri net.
- Arrival rate: the rate at which tokens arrive to the place per time unit.
- Waiting time: the time that passes from the (full) enabling of a transition until its firing, i.e. time that a token spends in the place waiting for a transition (to which the place is an input place) to fire and consume the token.
- Synchronization time: the time that passes from the partial enabling of a transition (i.e. at least one input place marked) until full enabling (i.e. all input places are marked). Time that a token spends in a place, waiting for the transition (to which this place is an input place) to be fully enabled.
- Sojourn time: the total time a token spends in a place during a visit (Waiting time + Synchronization time).
- Probabilities at XOR-splits: The probability that a case chooses a certain branch at a place with multiple outgoing arcs.

Two-transitions metrics

For each two (visible) transitions in the Petri net, the following metrics are available:

- Frequency: the number of process instances in which both transitions fire at least once.
- Time in between: (absolute) time between the first firing of the one transition during log replay and the first firing of the other transition.

Activity metrics

Often transitions are part of an activity, i.e. a task. For instance, an activity *clean* can be represented by the transitions *clean*-schedule, *clean*-start and *clean*-complete. In such case, activity metrics can be derived. These are:

- Arrival rate: rate at which work-items arrive at the activity.
- Waiting time: the time between the moment at which the activity is scheduled and the moment at which execution of the activity is started. (Time between a schedule and a start event of the activity).
- Execution time: the time in which an activity is actually executed. Which is the time between the moment at which the activity is started and the time at which it is completed, without possible time spend in a state of suspension. (Time between a start and a complete event of an activity, without the time spend in between all suspend and resume pairs that occurred in between).
- Sojourn time: Time between the scheduling of an activity and the time it finishes execution (Time between a schedule and a complete event).

2.3 CORRELATING BUSINESS PROCESS CHARACTERISTICS

Process discovery techniques make it possible to automatically derive process models from event data. However, often one is not only interested in discovering the control-flow but also in answering questions like "What do the cases that are late have in common?", "What characterizes the workers that skip this check activity?", and "Do people work faster if they have more work?", etc. Such questions can be answered by combining process mining with classification (e.g., decision tree analysis) [21].
The framework presented in [21] tries to discover correlations of different process characteristics. These characteristics can be based on the control-flow (e.g., the next activity going to be performed), the data-flow (e.g., the amount of money involved), the time perspective (e.g., the activity duration or the remaining time to the end of the process), the organization perspective (e.g., the resource going to perform a particular activity), or, in case a normative process model exists, the conformance perspective (e.g., the skipping of a mandatory activity). The general framework can be seen in Figure 4. The framework aims to support so-called analysis use cases. The definition of an analysis

use case is found in Definition 8, where $\mathcal{C}$ is the universe of process characteristics and where E is the universe of activities.

DEFINITION 8 (ANALYSIS USE CASE) *An analysis use case is a triple*$(\mathcal{C}_d, c_r, F)$ *consisting of*

- *a dependent characteristic* $c_r \in \mathcal{C} \setminus \mathcal{C}_d$,
- *a set* $\mathcal{C}_d \subset \mathcal{C}$ *of independent characteristics,*
- *an event-selection filter* $F \subset E$, *which characterizes the events that are retained for the analysis.*

The ultimate goal of the framework is to mine decision trees that explain the value of one characteristic, the dependent characteristic, in terms of the other characteristics, the independent characteristics. The decision trees are build based on an event log and an analysis use case. Decision trees classify instances (in this case events) by sorting them down in a tree from the root to some leaf node. Each non-leaf node specifies a test of some attribute (in this case, an independent characteristic) and each branch descending from that node corresponds to a range of possible values for this attribute. Each leaf node is associated to a value of a class attribute (in this case, the dependent characteristic). A path from root to a leaf represents a classification rule.

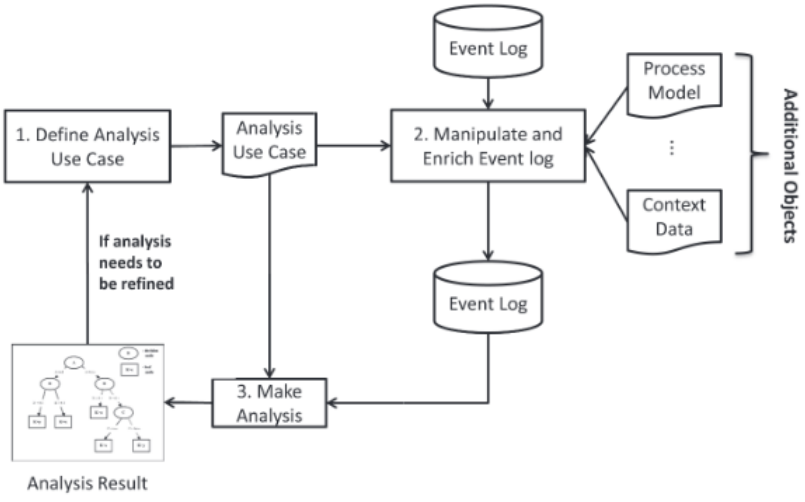

Figure 4: The general framework proposed in this paper: based on an analysis use case the event log is preprocessed and used as input for classification. Based on the analysis result, the use case can be adapted to gather additional insights [21].

The framework presented in [21] has been implemented as a plug-in in ProM named "Perform Predictions of Business Process Features"

and is used in the case study of this thesis. The plug-in allows for the user to configure a number of parameters, such as the level of decision tree pruning, the minimum number of instances per leaf or the discretization method. In this way, the user can try several configurations, thus, e.g., balancing between over- and under-fitting. Underfitting means that the derived decision tree allows for more behavior than actually recorded in the log, but results in the decision tree being smaller and easier to comprehend. Overfitting means that the derived decision tree specifies behavior recorded in the log very accurately, which can lead to large decision trees that are difficult to read. In the case study of this thesis the option case-level abstraction is used in the analysis use cases. Case-level abstraction replaces all the events in the trace with two events, the case-start and case-complete event. The case-start event is associated with the same values of the characteristics as the first event of the trace. The case-end event is associated with the last recorded values for all characteristics. For both events, the value of the Activity characteristic is overwritten with value "Case". To illustrate by example, applying case-level manipulation on the event log in Figure 5 results in the event log in Figure 6. As a result all characteristics are linked to a specific trace instead of events, allowing for an analysis looking at case level.

Case	Timestamp	Activity	Resource	Cost	NextActivityInTrace	ElapsedTime
1	1-12-2011:11.00	Preoperative Screening	Giuseppe	350	Laparoscopic Gastrectomy	0 days
1	2-12-2011:15.00	Laparoscopic Gastrectomy	Simon	500	Nursing	1.16 days
1	2-12-2011:16.00	Nursing	Clare	250	Laparoscopic Gastrectomy	1.20 days
1	3-12-2011:13.00	Laparoscopic Gastrectomy	Paul	500	Nursing	2.08 days
1	3-12-2011:15.00	Nursing	Andrew	250	First Hospital Admission	2.16 days
1	4-12-2011:9.00	First Hospital Admission	Victor	90	⊥	3.92 days
2	7-12-2011:10.00	First Hospital Admission	Jane	90	Laparoscopic Gastrectomy	0 days
2	8-12-2011:13.00	Laparoscopic Gastrectomy	Giulia	500	Nursing	1.08 days
2	9-12-2011:16.00	Nursing	Paul	250	⊥	2.16
3	6-12-2011:14.00	First Hospital Admission	Gianluca	90	Preoperative Screening	0 days
3	8-12-2011:13.00	Preoperative Screening	Robert	350	Preoperative Screening	1.96 days
3	10-12-2011:16.00	Preoperative Screening	Giuseppe	350	Laparoscopic Gastrectomy	4.08 days
3	13-12-2011:11.00	Laparoscopic Gastrectomy	Simon	500	First Hospital Admission	6.88 days
3	13-12-2011:16.00	First Hospital Admission	Jane	90	⊥	7.02 days
4	7-12-2011:15.00	First Hospital Admission	Carol	90	Preoperative Screening	0 days
4	9-12-2011:7.00	Preoperative Screening	Susanne	350	Laparoscopic Gastrectomy	0.66 days
4	13-12-2011:11.00	Laparoscopic Gastrectomy	Simon	500	Nursing	5.84 days
4	13-12-2011:13.00	Nursing	Clare	250	Nursing	5.92 days
4	13-12-2011:19.00	Nursing	Vivianne	250	⊥	6.16 days

Figure 5: Fragment of a hospital's event log with four traces [21].

Case	Timestamp	Activity	Resource	Cost	NextActivityInTrace	ElapsedTime
1	1-12-2011:11.00	Case	Giuseppe	350	Laparoscopic Gastrectomy	0 days
1	4-12-2011:9.00	Case	Victor	90	⊥	3.92 days
2	7-12-2011:10.00	Case	Jane	90	Laparoscopic Gastrectomy	0 days
2	9-12-2011:16.00	Case	Paul	250	⊥	2.16 days
3	6-12-2011:14.00	Case	Gianluca	90	Preoperative Screening	0 days
3	13-12-2011:16.00	Case	Jane	90	⊥	7.02 days
4	7-12-2011:15.00	Case	Carol	90	Preoperative Screening	0 days
4	13-12-2011:19.00	Case	Vivianne	250	⊥	6.16 days

Figure 6: The results after applying the case-level manipulation to the event log shown in Figure 5 [21].

Usage of the plug-in is explained in [17] and multiple example analysis use cases are described in [21]. To provide an example, the

following question Q1 "Are customer characteristics linked to the occurrence of reclamations? And if so, which characteristics are most prominent?" is taken from [21]. To answer this question, analysis use case U1 (depicted below) is defined and performed.

U1. ARE CUSTOMER CHARACTERISTICS LINKED TO THE OCCURRENCE OF RECLAMATIONS? *We aim to correlate the number of executions of activity Reclamation to the customer characteristics. We are interested in all decision-tree paths that lead to a number of executions of activity Reclamation greater than 0.*
Dependent Characteristic: *Number of Executions of Activity Reclamation.*
Independent Characteristics: *All characteristics of the events in the original log that refer to customers properties.*
Event Filter: *Every case-complete event is retained.*
Trace Manipulation: *Number of executions of Activity Reclamation, Case-Level Abstraction.[21]*

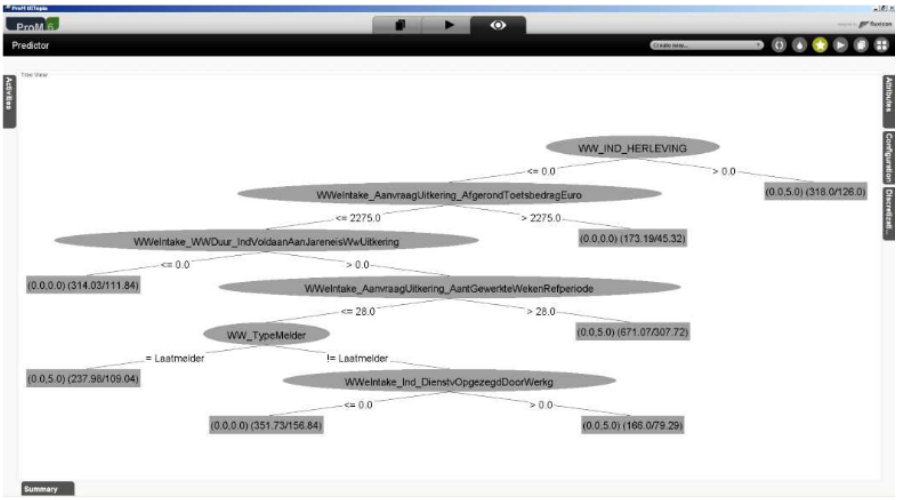

Figure 7: A screenshot of the framework in Figure 4 implementation in ProM that shows the decision tree used to answer question Q1 [21].

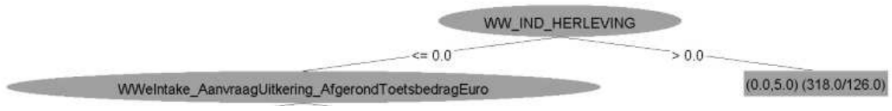

Figure 8: A close up of the root of the decision tree in Figure 7.

The results of performing this analysis use case are presented through the decision tree in Figure 7. In particular, Figure 7 refers to the configuration in which the minimum number of instances per leaf is set to 100 and the number of executions of Reclamation is set as the dependent characteristic and is discretized as two values: (0.0,0.0) and

(0.0,5.0). When the number of executions of Reclamation is 0, this is shown as (0.0,0.0) conversely, any value greater than 0 for the number of executions is discretized as (0.0,5.0). A close up of the root of this decision tree can be seen in Figure 8. A possible business rule that can be derived from Figure 8 is that if the customer is a recurrent customer (WW_IND_HERLEVING > 0), a reclamation occurs, i.e. the leaf is labelled as (0.0,5.0). The label is also annotated with 318.0/126.0, which indicates that a reclamation is not opened for 126 out of the 318 recurrent customers.

2.4 DECLARE

Over the last years, the declarative process modelling approach has flanked the classical procedural one [32], [25]. Declarative approaches only depict the behavioural constraints under which a process instance can unfold in its execution: as long as the constraints are not violated, the process instance is considered as valid. The declarative approach is a complementary strategy to the procedural models, which specify what are the next allowed activities at each stage of the process execution. Declarative process models are effective in a context of high flexibility for business processes [27]. The reason intuitively lies in the fact that fewer constraints allow for more possible executions. DECLARE [27] is a declarative process modelling language. It specifies an extensible set of constraint templates that are parametric with respect to the process activities. Declare is a declarative language that combines a formal semantics grounded in LTL on finite traces, with a graphical representation. In essence, a Declare model is a collection of LTL rules, each capturing a control-flow dependency between two activities [11].
Examples of DECLARE constraints are *Init*(a), and *Response*(b, c). The first one states that every trace must start with the execution of activity a. The second constraint imposes that if activity b is performed, then c must be performed eventually in the future. In Declare two main distinctions can be made between constraints, namely existence constraints and relation constraints. Existence constraints constrain one activity, whereas relation constraints constrain interplay of two activities. *Init* is an existence constraint as it constrains the execution of one activity in a process instances. *Response* is a relation constraint instead, because it constrains the interplay of two activities. Among the pair of constrained activities, there always are at least an *activation* and a *target*. The activation is an event whose occurrence constrains the possibility of other events (targets) to occur before or afterwards. For example, for the constraint 'every request is eventually acknowledged', each request is an activation. This activation is eventually associated with either a *fulfillment* or a *violation*, depending on whether or not the activation is matched with a target event that satisfies the

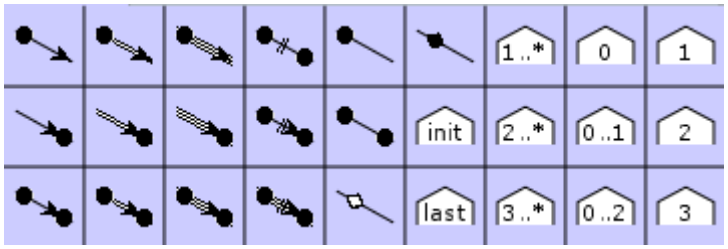

Figure 9: Graphical representation of every Declare constraint:
Top row (from left to right): response, alternate response, chain response, not co-existence, responded existence, exclusive choice, at least 1, absence, exactly 1.
Middle row (from left to right): precedence, alternate precedence, chain precedence, not succession, co-existence, initial task, at least 2, at most 1, exactly 2.
Bottom row (from left to right): succession, alternate succession, chain succession, not chain succesion, choice, last task, at least 3, at most 2, exactly 3.

constraint. Using again the example of requests that need acknowledgements, if the request occurs, this activation is associated with a fulfillment if the acknowledgement event is later observed; otherwise, the activation is associated with a violation. For *Response*(b, c), b is the activation and c is the target. The full list of DECLARE constraint templates can be found in [27]. Figure 9 shows the graphical representation of every possible constraint.

3

UWV CASE STUDY

This chapter has been removed due to confidentiality.

4

MINERFUL DECLARE MINER

In Chapter 3 a case study regarding the WIA Claim process is done by using a procedural approach. To determine whether a declarative analysis for the WIA Claim process yields useful results, a declarative process model needs to be drawn. This can be done by first mining an initial Declare model using a solid Declare miner.

This chapter contains the results of the MINERful Declare miner which is implemented as a ProM plug-in for this thesis. To be able to mine a Declare model that can be used for the declarative part of the UWV case study, an existing implementation of the MINERful algorithm is integrated in ProM as a plug-in named "MINERful Declare Miner". Implementing this existing implementation allows for a user friendly user interface for setting and tweaking parameters, a visual representation of the mined Declare model and easy cooperation with other plug-ins in the repertoire of ProM (i.e. the mined Declare model can be used for conformance checking, bottleneck analysis, etc.).
Section 4.1 gives an overview of MINERful algorithm and addresses its advantages over other declarative process discovery algorithms. Section 4.2 elaborates the user definable parameters of the plug-in and their effect on the mined models. Section 4.3 demonstrates the usage of the MINERful plug-in using a real life event log. Finally Section 4.4 concludes the chapter and gives a short elaboration on the maturity of the MINERful plug-in.

4.1 OVERVIEW MINERFUL DECLARE MINER

Artful processes [6] are a class of knowledge-intensive processes where the decisions taken during the enactment of the process are usually fast and based on the expertise and intuition of the main actors [3].

An example of an artful process is the management of a research project: knowledge workers such as project managers, professors, or technical managers contribute to the final outcome of the project. To this extent, they bring into play their competence together with the best practices gathered during their respective careers. Due to their nature, artful processes are rarely formalized [6]. Even though these artful processes are frequently repeated, they are not exactly reproducible (even by their originators) and can therefore not be easily shared either. Furthermore, hardly any process management systems

are currently used during the execution of such workflows [4].

The MINERful algorithm allows for the automated discovery of declarative control-flows for these artful processes. MINERful is a two-step algorithm. The algorithm accepts as input a set of traces T and an alphabet Σ. It requires that the characters of the traces (strings) in T belong to the alphabet Σ. In the first step a knowledge base is build, where statistical information extracted from logs is represented. This knowledge base is called *MINERfulKB* ($\mathcal{KB}$). *MINERfulKB* is build by using the two functions *computeKBOnwards* and *computeKBBackwards*. Function *computeKBBackwards* is the analog of *computeKBOnwards*. The latter reads the input strings from left to right, whereas the former parses the strings from right to left. In the second step, queries are evaluated on that knowledge base, in order to infer the constraints that constitute the discovered process using the function *discoverConstraints*. As such the second step infers the declarative model through the analysis of MINERfulKB. To influence the constraints that are discovered, parameters can be specified. These parameters, that are user definable, as well as their effect on the constraints are elaborated in Section 4.2. The final output is a set of constraints $\mathcal{B}^+$, verified on the knowledge base. In [4] detailed descriptions are present of the construction of MINERfulKB and how the analysis on this knowledge base works. An overview of the MINERful algorithm can be seen in Algorithm 1.

Algorithm 1 The MINERful pseudo-code algorithm, in its simplest form (bird's eye view)

1: $\mathcal{KB} \leftarrow$ *computeKBOnwards(*T, Σ, $\emptyset$*)*
2: $\mathcal{KB} \leftarrow$ *computeKBBackwards(*T, Σ, $\mathcal{KB}$*)*
3: $\mathcal{B}^+ \leftarrow$ *discoverConstraints(*$\mathcal{KB}$, Σ, $|\mathsf{T}|$*)*
4: return $\mathcal{B}^+$

DECLARE is the declarative process modeling language used in this thesis. A list of the constraint templates used in the remainder of this thesis are listed in Table 2, with `a`, `b`, and `c` as example activities. The full list of existing DECLARE constraint templates can be found in [27].

This chapter reports on the implementation of MINERful in ProM, a technique to mine declare process models from an existing event log [4]. Since DECLARE is the used declarative modelling language in this thesis, DECLARE is implemented to be the default modelling language of the mined models. Compared with other existing techniques, MINERful has shown the best scalability with respect to the input size, in terms of number of traces, length of traces and activities of the process. Readers are referred to [4] for more details about this comparison.

CONSTRAINTS	DESCRIPTION
Init(a)	a should be the first activity in a trace
AtMostOne(a)	a should be executed at most once
Participation(a)	a should be executed at least once
RespondedExistence(a,b)	If activity a is executed, b also has to be executed either before or after a
CoExistence(a,b)	If one of the activities a or b is executed, the other one also has to be executed
NotCoExistence(a,b)	If one of the activities a or b is executed, the other is never executed
Response(a,b)	When a is executed, b has to be executed after a
AlternateResponse(a,b)	When a is executed, b has to be executed after a and no other a can be executed in between
ChainResponse(a,b)	a is immediately followed by b
Precedence(a,b)	b has to be preceded by a
AlternatePrecedence(a,b)	b has to be preceded by a and another b cannot be executed between a and b
ChainPrecedence(a,b)	b is immediately preceded by a
NotPrecedence(a,b)	b cannot be preceded by a
Succession(a,b)	Combination of *Response*(a,b) and *Precedence*(a,b)
AlternateSuccession(a,b)	Combination of *AlternateResponse*(a,b) and *AlternatePrecedence*(a,b)
ChainSuccession(a,b)	a is immediately followed by b and b is immediately preceded by a
NotSuccession(a,b)	a is never followed by b and b is never preceded by a
NotChainSuccession(a,b)	a is not allowed to be immediately followed by b

Table 2: Table listing the constraints mentioned in the thesis.

As mentioned, the MINERful algorithm discovers declarative control-flows in the context of artful processes. As is elaborated in Chapter 3, actors are present in the WIA Claim process which identify the severity of health issues of the client (that in turn defines the outcome of the process). For this reason, as well as its great performance for large event logs, it was chosen to use the MINERful algorithm for the declarative part of the case study for this thesis.
The existing implementation of MINERful [7] utilized parameters input via command prompt to read event logs and produce models as xml files from and to folders. This method of mining models is exhaustive when searching for the desired parameters. As a result a new implementation is presented in this thesis which has been realised in ProM. To use the MINERful Declare Miner with ProM, it is necessary

to download the ProM Nightly build [23] and, subsequently, install the *DeclareMinerFul* package through ProM's Package Manager.

4.2 THE MINERFUL ALGORITHM PARAMETERS

The application of MINERful uses an event log as input. Examples computations will be elaborated using the following example log: $\{\langle a, b, a, c\rangle, \langle a, b, b, a, c, b, a\rangle, \langle a, c, c\rangle, \langle a, b, c\rangle\}$. The application of the MINERful plug-in for the DECLARE-model discovery can be customised through four parameters, namely:

SUPPORT. It is the number of fulfillments divided by either *(i)* the number of traces in the log, in the case of existence constraints like *Init*(a), or *(ii)* the number of occurrences of the activations (in the case of relation constraints like *Response*(b, c)). In the example log, the support of *Init*(a) is 1.0, because all traces start with a, whereas the support of *Response*(b, c) is 0.8, as 4 b's out of 5 fulfil the constraint.

CONFIDENCE. It is the product of the support and the fraction of traces in the log where either *(i)* the constrained activity occurs (existence constraints), or *(ii)* the activation occurs (relation constraints). The confidence of *Init*(a) is $1.0 \cdot 1.0 = 1.0$ and the confidence of *Response*(b, c) is $0.8 \cdot 0.75 = 0.6$, since b occurs in 3 traces out of 4.

INTEREST FACTOR. It is the product of confidence and the fraction of traces in the log where either *(i)* the constrained activity occurs (existence constraints), or *(ii)* the target occurs (relation constraints). The interest factor of *Init*(a) is $1.0 \cdot 1.0 \cdot 1.0 = 1.0$, and the interest factor of *Response*(b, c) is $0.8 \cdot 0.75 \cdot 1.0 = 0.6$, since c occurs in all traces.

SKIP NEGATIVE CONSTRAINTS. When the process is characterised by parts with a rigid structure, the discovered model may blow up in term of presence of negative constraints. Therefore, analysts are provided with an option to not considering negative constraints, thus increasing the readability of the discovered models.

The first three parameters estimate a level of relevance for a constraint, based on the assumption that the more the constrained activities appear in the log, the more their constraints should be taken into account. As such, if an activity a appears once in an event log containing hundreds of thousands of events, it is likely the execution of the task is wrong.

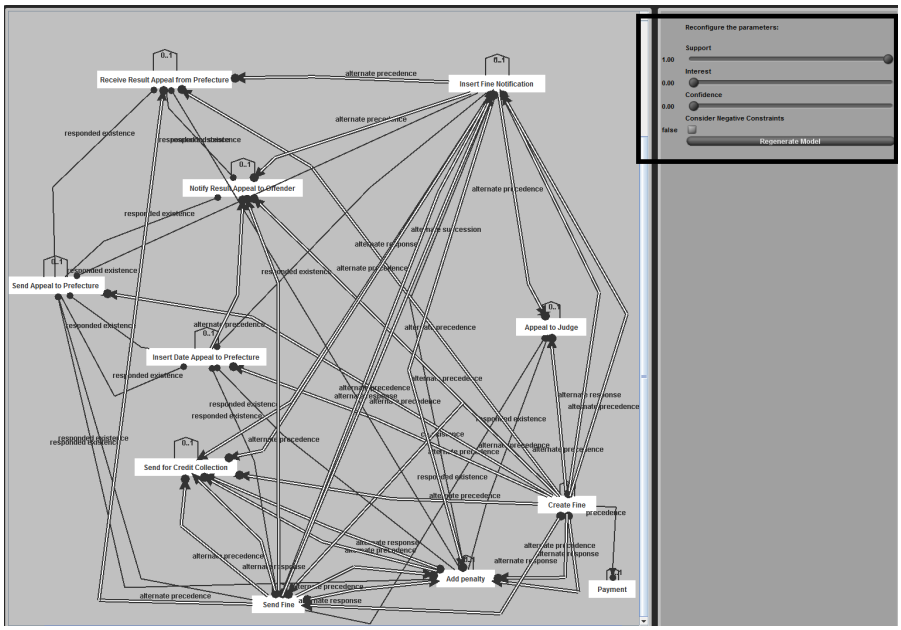

Figure 10: The resulting output screen from MINERful's where the model has been discovered by setting support to 0.5 and setting confidence and interest factor to 0.

4.3 USAGE OF THE TOOL ON A USE CASE

In this Section the functionalities of MINERful will be demonstrated using the publicly available real-life event log Road Traffic Fine Management Process [14]. In Chapter 6 the technique is applied to the event log extracted for the UWV case study. The Road Traffic Fine Management Process event log records executions of instances of the process enacted in an Italian local police office for managing fines for road traffic violations. It contains 150,370 traces and 561,470 events for 11 different process activities.

Initially, the MINERful plug-in was executed skipping the negative constraints and using the following values for the other parameters: *(i)* support $= 0.50$, *(ii)* confidence $= 0.00$, *(iii)* interest factor $= 0.00$. The resulting declarative process model can be seen in Figure 10.

The output view consists of two panels. The panel on the left-hand side contains the mined declarative process model. The user is free to relocate activities and constraints to manually improve the readability. The panel on the right-hand side allows the user to adjust the four parameters mentioned in Section 4.2 (see the area delimited by a black rectangle in Figure 10) instead of restarting the plugin. After clicking the *Regenerate Model* button, the left panel will update the view with the newly mined declarative process model after some computation time.

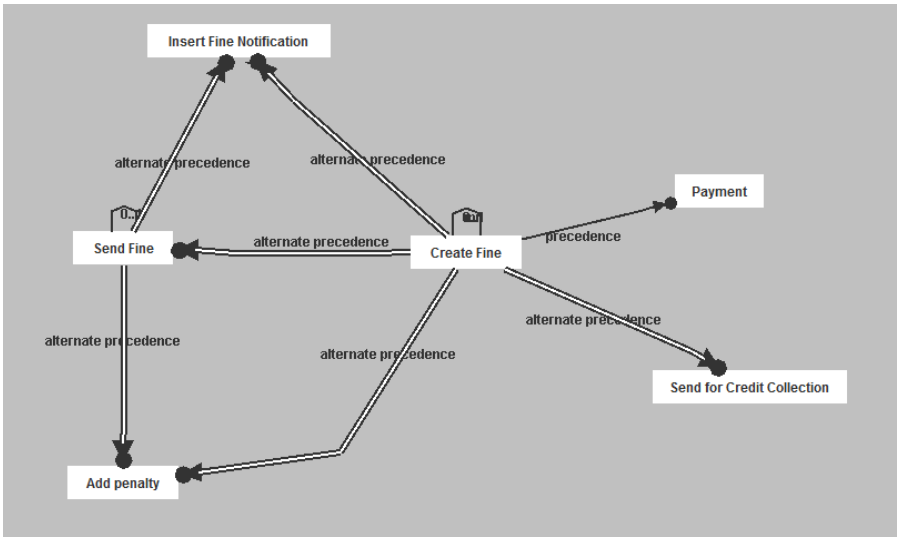

Figure 11: The resulting output screen from MINERful's example with stricter parameters (i.e., closer to 1).

The model in Figure 10 has been obtained by assigning value 0 to all parameters, except for support. This configuration has produced a cluttered declarative process model with many constraints, i.e. the model is probably overfitting the event log. The increase of the value of any parameter would generate a model with fewer constraints, thus probably reducing the overfitting problems and, also, improving the readability of the declarative process model. Of course, an excessive increase of any parameter may have a detrimental effect on the precision of the discovered model: the model may underfit the event log, allowing for too much behaviour. The declarative process model shown in Figure 11 derives from the application of the following parameters: *(i)* support $= 0.70$, *(ii)* confidence $= 0.30$, *(iii)* interest factor $= 0.00$.

4.4 CONCLUSION

This chapter presented a new ProM plug-in named "MINERful Declare Miner" that allows for the automated discovery of declarative control-flows for artful processes. The new implementation presents a user friendly user interface for setting and tweaking parameters and a visual representation of the mined Declare model. As a result similar declarative case studies, namely case studies regarding processes with decision making during the enactment of the process, in the future can utilize this miner to produce a suitable Declare model. The process models in this chapter were discovered using a laptop equipped with a Intel Core i3 with 4GB of RAM. With this modest hardware, the MINERful plug-in was able to mine the model in less than 30 seconds using a real-size event log with 561,470 events belong-

ing to 150,370 traces. ProM currently contains two declarative process model discovery plug-ins, the "Declare Maps Miner" presented in [10] and the "Data-Aware Declare Miner" presented in [11]. The "Declare Maps Miner" was not able to mine a model for the same event log after hours of mining. The current implementation of this miner is unstable and is not able to mine a model when given a large event log as input. The "Data-Aware Declare Miner" mines empty models, i.e. a model with 0 tasks and 0 constraints, at the moment of writing this thesis. This indicates that the MINERful plug-in has reached a large degree of maturity as it performs extremely well in terms of scalability. The scalability of the technique and its implementation allows its application for the declarative part of the UWV case study where the log is very large.
Also, the plug-in is integrated with the entire repertoire of techniques that are already available in ProM (see, e.g., [10]): the mined Declare model can be later used for conformance checking, bottleneck analysis, improvement, etc.

In cooperation with Claudio Di Ciccio, Massimiliano de Leoni and Jan Mendling the contents of this chapter have been accepted as Demo paper to the BPM 2015 Demo Track [5].

5

DATA-AWARE COMPLIANCE CHECKING OF DECLARE MODELS

Chapter 4 presented a technique that can mine a Declare model that is suitable to the WIA Claim process. Now such an initial model can be mined, data-aware compliance checking techniques need to be available to determine whether a declarative analysis of the WIA Claim process yields useful results. The compliance checking technique needs to be data-aware as multiple decision points are present in the process that are governed by conditions.

This chapter contains the results of a data-aware compliance checking technique that is implemented as a ProM plug-in for this thesis. At the moment existing conformance checking techniques for Declare models are limited to only checking the control-flow. This chapter presents a compliance checking technique that takes data into account. The thesis limits to a compliance checking technique that determines for each constraint in the Declare model the set of traces that violate the constraint, i.e. the technique does not determine optimal alignments for each trace like a conformance checking technique does. Section 5.1 explains the algorithm that is developed to perform data-aware compliance checking for a given Declare model and event log. Section 5.2 addresses the implementation and illustrates the usage of the constructed plug-ins in ProM. Section 5.3 validates the correctness of the implementation using synthetic event logs. Finally Section 5.4 addresses the conclusions.

5.1 DATA-AWARE COMPLIANCE CHECKING ALGORITHM

This section presents the algorithm that is used to apply data-aware compliance checking for Declare models enriched with guards.

Before data-aware compliance checking can be done for Declare models, first guards need to be specified for Declare models. In existing data-aware conformance checking techniques for procedural approaches, guards are placed on the task itself. In [11] a Declare miner is presented that mines Declare models that possibly have guards. The guards in this approach are placed on the constraints. In this thesis a similar approach is chosen, i.e. the guards are placed on the constraints. Recall that among a pair of constrained activities, there always are at least an *activation* and a *target*. Following what is proposed in [11] the guard is evaluated at the moment the constraint is

activated, i.e. the moment activity *activation* occurs. In case a guard is evaluated as *true*, the target should be fulfilled. In case a guard is evaluated as *false*, the constraint is considered non-existent and as a result should not be fulfilled. The thesis limits to evaluating relation constraints, i.e. existence constraints are not considered.

To illustrate by example, the following event log is used: $\{\langle A, A, B\rangle, \langle A, A, C\rangle\}$, where for both traces is specified in the event log that the *Age = 40*. Let Constraint 1 be *Response*$(A, B, \mathtt{Age} > 30)$ and let Constraint 2 be *Precedence*$(A, B, \mathtt{Age} < 30)$. Both traces in the example event log trigger Constraint 1 since the age is larger than 30. Both traces do not trigger Constraint 2 since the age is not smaller than 30, as a result the constraint is considered to be non-existent. The first trace in the example event log fulfills both constraints: Constraint 1 since A is followed by B and Constraint 2 for above mentioned reasons. The second trace in the example event log does not fulfill both constraints: Constraint 1 is violated since A is not followed by B and Constraint 2 is fulfilled for above mentioned reasons.

With evaluation of guards specified, the data-aware compliance checking algorithm can be specified. The algorithm takes as input a Declare model $\mathcal{D} = (A, \Pi)$ and an event log L, where A is the set of activities and Π is the set of constraints defined over activities in A. For compliance checking, it is unnecessary to distinguish the activities in the log that do not appear in A. An activity that does not appear in A but does appear in the log is referred to as $\checkmark$. The goal of the algorithm is to determine for each constraint $\pi \in \Pi$ the set of traces $\mathcal{L}$ that violate π. In [28] a technique is presented that translates Declare constraints as final-state automata. Inspired by this technique, in this thesis it is checked whether a trace σ is compliant with a Declare constraint π by translating π into a final-state automaton that accepts all traces that do not violate π. The final-state automatons in this thesis are made such that guards are taken into account and are referred to as *constraint automatons*.

DEFINITION 9 (CONSTRAINT AUTOMATON) *Let $\mathcal{D} = (A, \Pi)$ be a Declare model, $\pi \in \Pi$ and $\Sigma = A \cup \{\checkmark\}$. The constraint automaton $\mathcal{A}_\pi = (\Sigma, \Phi, \Psi_\pi, \psi_{0_\pi}, \delta_\pi, F_\pi)$ is the final-state automaton which accepts precisely those traces $\sigma \in \Sigma^*$ satisfying π, where:*

- $\Sigma = A \cup \{\checkmark\}$ *is the input alphabet;*
- Φ *is the universe of linear inequations and boolean operators AND, OR and NOT*
- Ψ_π *is a finite, non-empty set of states;*
- $\psi_{0_\pi} \in \Psi_\pi$ *is an initial state;*

- $\delta_\pi \in \Psi_\pi \times \Sigma \times \Phi \to \Psi_\pi$ *is the state-transition function;*
- $F_\pi \subseteq \Psi_\pi$ *is the set of final states. [18]*

A constraint automaton A_π is constructed for every constraint $\pi \in \Pi$ and is initially in state ψ_{0_π}. Every trace $\sigma \in L$ is replayed in A_π to determine whether A_π ends in a so called final-state. Replaying a trace σ is done by sequentially processing each activity e in σ. If the activity name of e is accepted by a state-transition function, the constraint automaton follows the state-transition to a next state. In case the next activity in σ to be processed in automaton A_π does not allow for a valid state-transition, the automaton deadlocks and can never lead to a final-state. Constraints with no guard specified are considered as constraints with guards where the condition holds. In case a constraint automaton stops in a final-state when σ is processed entirely, the constraint is fulfilled. In this thesis, a semantics has been defined for data-aware constraints in terms of final-state automatons. The full list of automatons corresponding to the constraints listed in [27] can be seen in Appendix C. Automatons for Co-Existence and Succession constraints have not been constructed, since their functionalities can be matched by combining two Responded Existence automatons and combining a Response- and Precedence automaton respectively.

Examples 1 and 2 give an example of the constraint automatons corresponding to the *Response*(A, B, Cond) and *NotPrecedence*(A, B, Cond) constraints respectively.

EXAMPLE 1 (RESPONSE) *The automaton for the Response(A, B, Cond) constraint can be seen in Figure 12. State 0 is the initial state of the automaton, depicted by the incoming arrow originating from a black dot instead of a different state. State 0 is also a final-state, depicted by the double outline. A transition is labeled with the set of the activities triggering it, where* Σ *is the entire input alphabet. At the moment A occurs and* Cond *holds, B should occur. This achieved by making a distinction between A where* Cond *holds and A where* Cond *does not hold. The occurrence of any task beside A where* Cond *holds results in the automaton staying in the initial state, i.e. the constraint is not triggered. The occurrence of A where* Cond *holds results in the automaton going to a non final-state state 1, where the state is maintained for all tasks beside target B. At the moment B occurs, the automaton goes back to the initial state (which is a final-state) as the constraint is fulfilled. The occurrence of another A where* Cond *holds, restarts the process.*

EXAMPLE 2 (NOT PRECEDENCE) *The automaton for the NotPrecedence(A, B, Cond) constraint can be seen in Figure 13. State 0 is the initial state of the automaton and is also a final-state. At the moment activation B occurs and* Cond *holds, a task A should not have occurred in the*

past. This is achieved by 'blocking' the occurrence of B *where* Cond *holds, at the moment* A *has occurred. The occurrence of any task beside* A *results in the automaton staying in the initial state. The occurrence of target* A *results in the automaton going to a final-state state 1. The automaton remains in the same state for all tasks beside* B *where* Cond *holds. Since* B *where* Cond *holds is not specified as a transition from state 1, an occurrence will be considered as faulty.*

Beside checking for each trace $\sigma \in L$ whether it ends in a final-state of constraint automaton A_π, it is checked whether the *activation* with a guard that holds is present in the trace. Traces containing the *activation* with a guard that holds are called *activated traces*. Using this fact, the fulfillment ratio of A_π is calculated as $1 - \frac{\#failingtraces}{\#activatedtraces}$. It was chosen to use the number of activated traces over the total number of traces, since traces where the constraint is never triggered are not interesting to the statistics.

Algorithm 2 contains the pseudocode giving a step by step illustration of the algorithm. Function *returnCorrespondingAutomaton* constructs a constraint automaton for the given constraint and its possible guard. ψ_π is used to represent the current state of the final-state automaton. Given an event e, function $act(e)$ returns the task name a of the event, where $a \in A$. Function $dataAss(e)$ returns the data attribute values for every variable $v_e \in V$. With this information the variable assignment can be updated for the current event. The variable assignment is used to evaluate a condition in case it is specified on the constraint. As can be seen in Algorithm 2, the data attribute values are read for every event in the trace. As a result it possible that for a trace where task A is present more than once, the guard can be evaluated as $true$ and as $false$ (or vice versa) at some point during processing the trace if the data value is variable in the log. This approach was chosen over the writing and reading of data values at fixed points in the model, as is currently done in procedural data-aware conformance checking techniques, as it better suits the free nature of the Declare language. Finally, function $activatedTraces(L, \pi)$ returns to the total number of *activated traces*.

As can be seen in Algorithm 2, the input event log is traversed once for every constraint in the input Declare model. For each additional constraint added to a Declare model, results in traversing the event log an additional time. From this follows the running time of the algorithm is linear.

Algorithm 2 Data-Aware Compliance Checking Algorithm

Input 1: Declare Model $\mathcal{D} = (A, \Pi)$
Input 2: Event Log L
for Relation Constraint $\pi \in \Pi$ **do**
    int failingTraces $\leftarrow 0$
    Automaton $\mathcal{A}_\pi = (\Sigma, \Phi, \Psi_\pi, \psi_{0_\pi}, \delta_\pi, F_\pi) \leftarrow$ returnCorrespondingAutomaton(π)
    for Trace $\sigma \in L$ **do**
        State $\psi_\pi \leftarrow \psi_{0_\pi}$
        boolean Stop $\leftarrow$ *false*
        Assignment $p \leftarrow \emptyset$
        for Event $e \in \sigma$ **do**
            for Variable $v \in dom(dataAss(e))$ **do**
                $p(v) \leftarrow varAss(v)$
            end for
            if not exists $\psi_\pi' = \delta_\pi(\psi_\pi, act(e), Cond)$ where Cond evaluates as *true* with assignment p **then**
                Stop $\leftarrow$ *true*
                break
            else
                $\psi_\pi \leftarrow \psi_\pi'$
            end if
        end for
        if Stop = true $\|$ $\psi_\pi \notin F_\pi$ **then**
            failingTraces++
        end if
    end for
    if $activatedTraces(L, \pi) == 0$ **then**
        fulfillmentRatio(π) $\leftarrow 1.00$
    else
        $$\text{fulfillmentRatio}(\pi) \leftarrow 1 - \frac{failingTraces}{activatedTraces(L, \pi)}$$
    end if
end for
Output: fulfillmentRatio

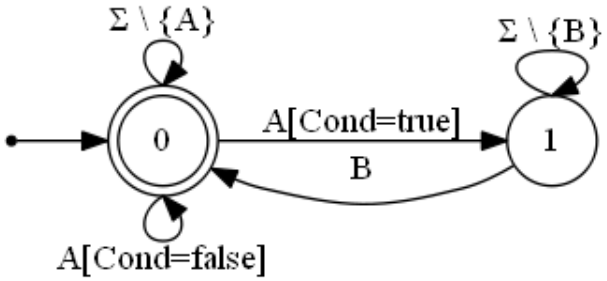

Figure 12: Constraint automaton for Response(A,B,Cond) - if A occurs and Cond holds, B must occur afterwards

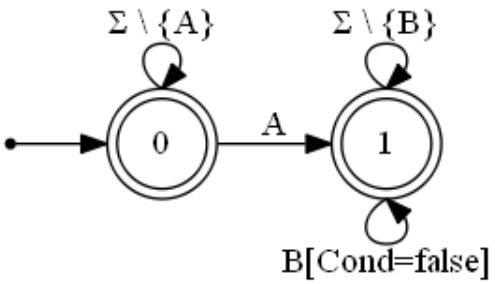

Figure 13: Constraint automaton for Not Precedence(A,B,Cond) - if B occurs and Cond holds, A cannot have occurred before

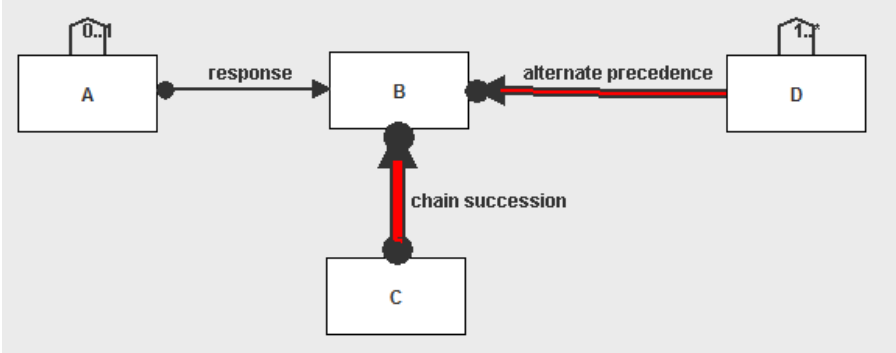

Figure 14: Example Declare model containing multiple activities and constraints

5.2 TOOL SUPPORT

The technique presented in Section 5.1 has been implemented as two separate plug-ins for ProM. The first plug-in is called "Create/Edit DeclareMap With Data", which allows for enriching or editing imported/mined Declare models with guards. The second plug-in is called "Declare Data-Aware Compliance Checker" which applies Algorithm 2 on a given Declare model and event log to determine the fulfillment ratio for each constraint in the Declare model. It is necessary to download the ProM Nightly build [23] and, subsequently, install the *DeclareChecker* package through ProM's Package Manager to access these plug-ins. Beside the implementation of these two new techniques, the "Perform Predictions of Business Process Features" plug-in has been enhanced with the functionality of taking the "Declare Data-Aware Compliance Checker" results into account.

To illustrate the usage of the two plug-ins the Declare model in Figure 14 is used in combination with the following example log: $\{\langle A, C, B\rangle, \langle D, C, B\rangle, \langle D, C, B\rangle, \langle A, C, D\rangle\}$. In this example log two variables are present: 'Age' and 'City'. The values for these variables are '50' and 'Eindhoven', '20' and 'Eindhoven', '20' and 'Utrecht' and '41' and 'Eindhoven' for the four traces respectively.

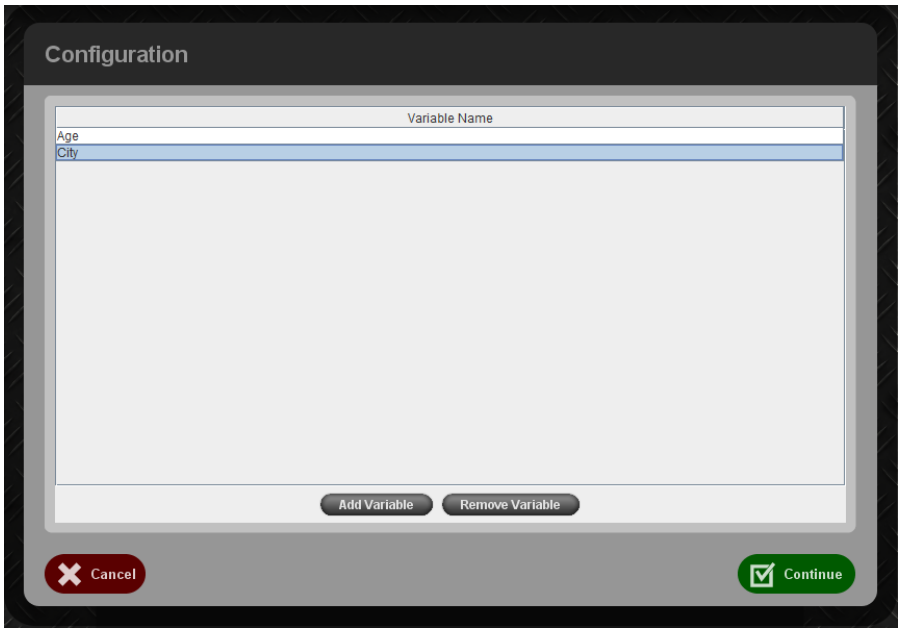

Figure 15: The first dialog screen in the Create/Edit DeclareMap With Data plug-in

5.2.1 *Create/Edit DeclareMap With Data*

The "Create/Edit DeclareMap With Data" plug-in takes a DeclareMap or DataDeclareMap file as input. The first filetype is the standardized Declare model type used by multiple Declare related plug-ins in the ProM framework, whereas the latter has been constructed for this thesis. To illustrate the usage of the plug-in, the DeclareMap in Figure 14 is taken as input. After selecting the plug-in, the user is presented the dialog screen in Figure 15. The user can use the button 'Add Variable' to extend the variable list with an extra field. In this field the user can fill in a variable desired to be present in one or more guards. The 'Remove Variable' removes variables from the specified list. In this example two variables are specified, namely 'Age' and 'City'. When creating a new DataDeclareMap, the list of variables is initially empty. When editing a DataDeclareMap the list of variables contains the variables that have been specified before. The 'Cancel' button exits the plugin. The 'Continue' button moves to the next dialog screen.

The second dialog screen can be seen in Figure 16. The second dialog screen lists all the relation constraints that are present in the input Declare model in the 'Relation' column. The activation and the target of each constraint have been added to be able to distinguish between constraints with the same name. Note that the plug-in does not support enriching existence templates with guards as they do not show up in the list. When creating a new DataDeclareMap, the col-

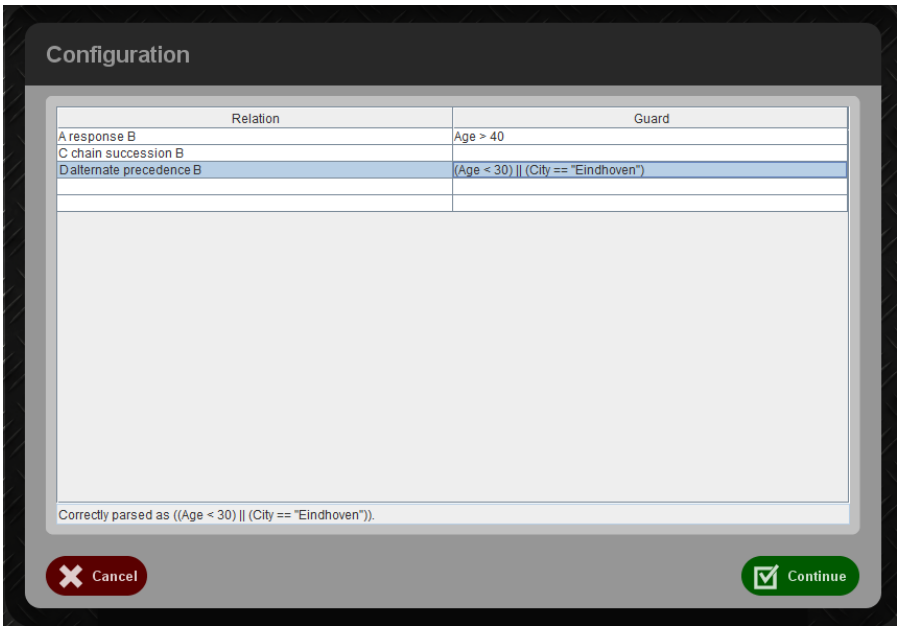

Figure 16: The second dialog screen in the Create/Edit DeclareMap With Data plug-in

umn 'Guard' is initially empty. When editing a DataDeclareMap the column 'Guard' contains the guards that have been specified before. The plug-in supports guards consisting of atoms of the form 'variable op constant' and 'variable op variable', where 'op' is a relational operator (e.g., =, <, or >). The constants can be strings, integers, doubles, dates or booleans. Logical operators can be used to apply multiple conditions in a guard. In this example two guards are specified, namely 'Age > 40' and '(Age < 30) || (City == "Eindhoven")'. As can be seen in Figure 16, the user gets feedback when the guard is correctly parsed. In case a variable is used which has not been specified in the previous dialog screen or in case parenthesis are not correctly used, the user receives a message the guard cannot be correctly parsed. The 'Cancel' button exits the plugin. The 'Continue' button moves to the output screen.

The new Declare model displayed in the output screen can be seen in Figure 17. The output screen of the plug-in shows the DeclareMap that has been enriched with the specified guards. Every constraint enriched with a guard, has a label of the constraint name combined with the guard. Note that the existence templates are still present in the output screen.

5.2.2 *Declare Data-Aware Compliance Checker*

The "Declare Data-Aware Compliance Checker" takes a DeclareMap or DataDeclareMap file and an event log as input. Logically when tak-

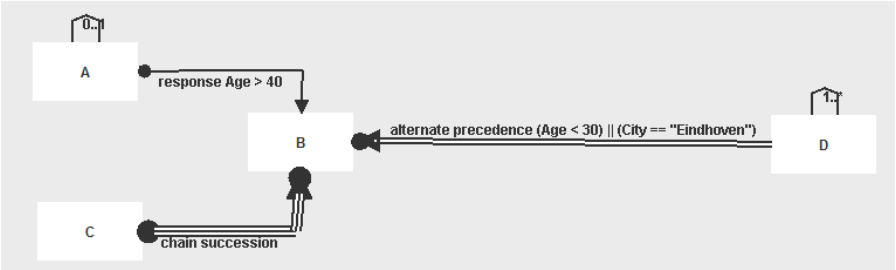

Figure 17: The output screen of the Create/Edit DeclareMap With Data plug-in

CONSTRAINT	# ACTIVATIONS	# VIOLATIONS	FULFILLMENT RATIO
A response B	2	1	0.5
C chain succession B	4	1	0.75
D alt precedence B	3	1	0.67

Table 3: Table listing the constraints of the Declare model in 14 with their corresponding activations, violations and their fulfillment ratio.

ing a DeclareMap as input no guards are present, i.e. a DeclareMap is a DataDeclareMap where all guards are true at all time. To illustrate the usage of the plug-in, the DataDeclareMap in Figure 17 is taken as input in combination with the earlier defined example log $\{\langle A, C, B\rangle, \langle D, C, B\rangle, \langle D, C, B\rangle, \langle A, C, D\rangle\}$. The plugin does not allow for manually mapping variables in the event log to variables in the DataDeclareMap. As a result the user is responsible for specifying variable names in the "Create/Edit DeclareMap With Data" plug-in that are identical to the variable names present in the log. After selecting the plug-in, the user is presented no dialog screen and is immediately presented the output screen shown in Figure 18. For each constraint the name of the constraint, the number of failing traces and the number of activated traces is printed in the command prompt. This way this information is easily accessible to the user of the plug-in.

The number of activations, number of violations and the fulfillment ratio for every constraint can be seen in Table 3. Figure 18 shows that the colors of each constraint are different graduations of red. The plug-in colors the constraints based on their fulfillment ratio. A fulfillment ratio of 1.0 results in a black color and a fulfillment ratio of 0.0 results in a white color. A fulfillment ration in between 0.0 and 1.0 results in different graduations of red. On the right hand side of the output screen a button label "Toggle Label View" can be brought forth. Clicking this button results in the Declare model in the output screen to be redrawn to a Declare model with the fulfillment ratio written

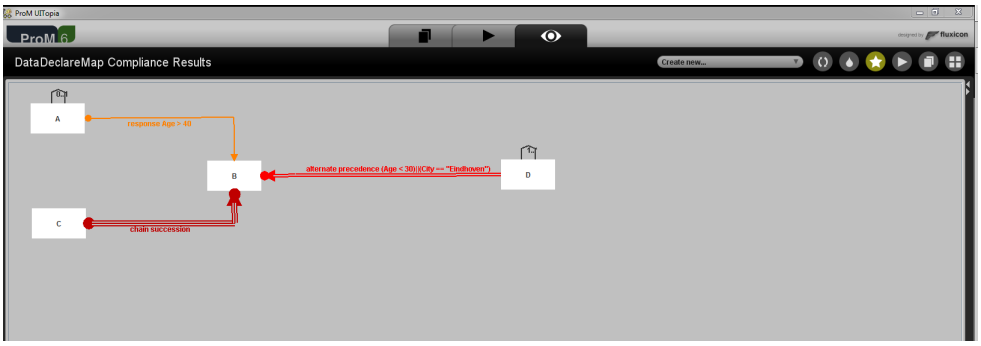

Figure 18: The initial output screen of the Declare Data-Aware Compliance Checker plug-in, where the name and the possible guard is displayed for every constraint

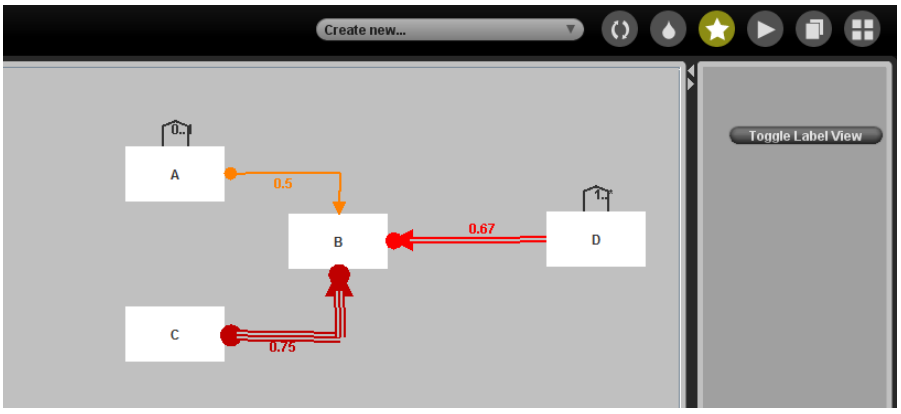

Figure 19: An alternative output screen of the Declare Data-Aware Compliance Checker plug-in, where the fulfillment ratio is displayed for every constraint

above the constraint. Applying this view on the example results in the Declare model shown in Figure 19.

As it is useful to know which traces violated of fulfilled a certain constraint, the user can right click on every constraint. This results in a menu appearing, illustrated for the example in Figure 20. By clicking one of the options the user can extract a sub-log with all traces fulfilling or violating the constraint, based on the option chosen. These event logs can immediately be used in ProM or exported for later usage. For technical reasons, trace names need to be unique. Clicking the "Extract violating traces" option in the example results in an event log containing the first trace of the example log: $\{\langle A, C, B\rangle\}$.

5.2.3 *Perform Predictions of Business Process Features*

In the procedural part of the case study, the data-aware conformance checking results are taken into account in the "Perform Predictions of Business Process Features" plug-in. Even though these conformance

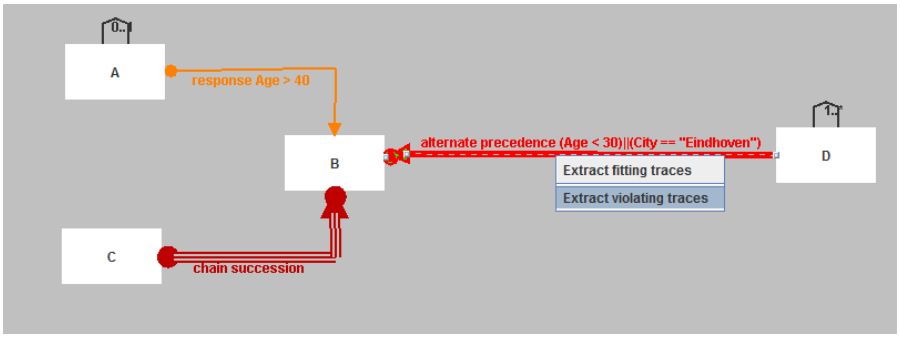

Figure 20: The menu appearing after right clicking a constraint in the Declare Data-Aware Compliance Checker plug-in

checking results were not decisive in the decision making and did not show up in the derived decision trees presented in Section **??**, considering these violations might yield useful results in other case studies. For this reason the "Perform Predictions of Business Process Features" plug-in has been updated to take the data-aware compliance checking results of the "Declare Data-Aware Compliance Checker" plug-in into account.

For every constraint π in the data-aware compliance checking results a variable is created named "Constraint *'activation* π *target'* violated", where *activation* and *target* are the activation and target of the constraint. These variables are added to every event in the given event log. The variables are given the string value "Yes" or "No", based on whether the trace the event belongs to is violated ("Yes") or fulfilled ("No"). Figure 21 shows the "Perform Predictions of Business Process Features" plug-in where data-aware compliance checking results of Figure 18 are taken into account. The user can select for each of the constraints whether he or she wants to consider them in the decision trees. A possible decision tree generated for the compliance checking results of constraint "D alternate precedence B" can be seen in Figure 22.

5.3 VALIDATION OF THE IMPLEMENTATION

To determine whether the implementation functions correctly, numerous test traces from created synthetic event logs are ran through every constraint. For each constraint type c present in Figure 9 a Declare model is constructed. These Declare models contain two tasks A and B which are constrained with the constraint "A c B".
The constraints are divided in four groups, where each group is tested using a synthetic event log. The groups are made since a specific synthetic event log can be used to test multiple constraints.

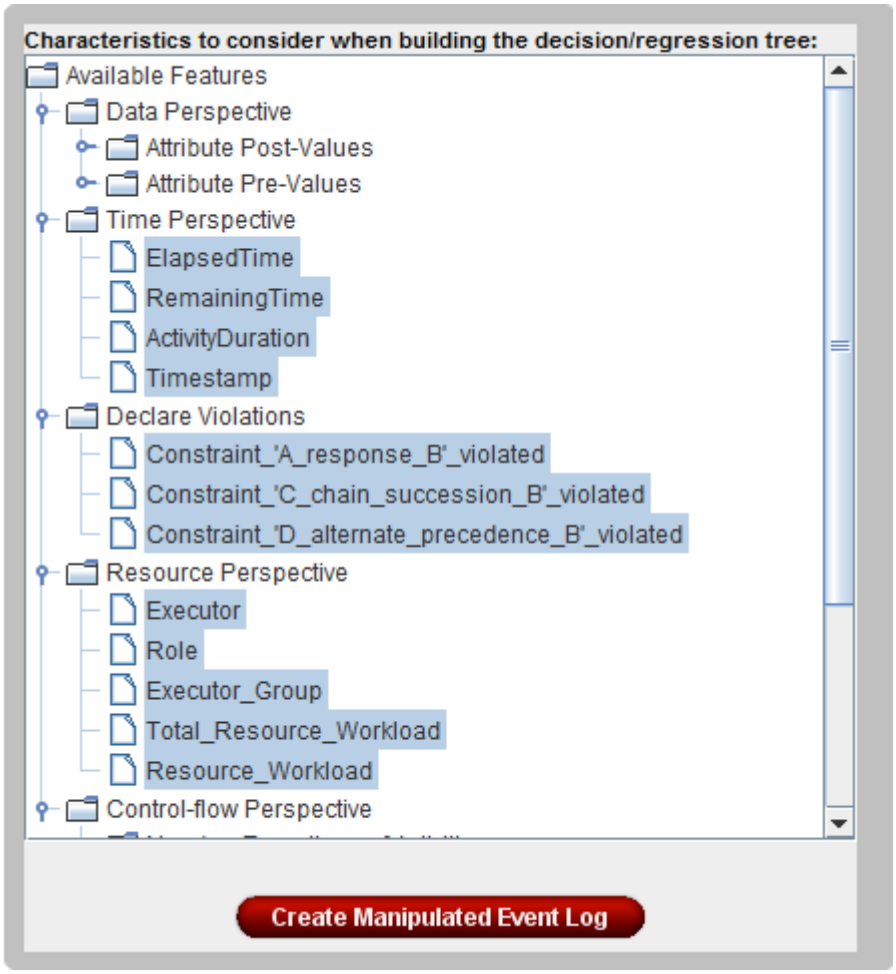

Figure 21: The extended Attributes panel in the "Perform Predictions of Business Process Features" plug-in, which now contains constraint violation information from the "Declare Data-Aware Compliance Checker" plug-in

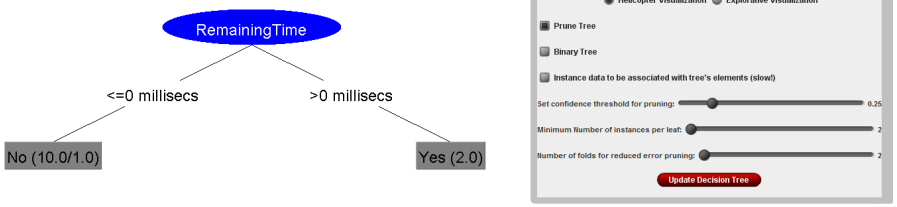

Figure 22: An example decision tree generated using constraint violation information from the "Declare Data-Aware Compliance Checker" plug-in

Group 1 contains constraints {*Response (R)*, *Alternate Response (AR)*, *Chain Response (CR)*, *Responded Existence(RE)*}. The results of testing these constraints can be seen in Table 4. The first column in this table

TRACE	R	AR	CR	RE
AAB {A,A = true}	✓	×	×	✓
AAB {A,A = false}	NA	NA	NA	NA
AAC {A,A = true}	×	×	×	×
AAC {A,A = false}	NA	NA	NA	NA
ABAB {A,A = true}	✓	✓	✓	✓
ACBAB {A,A = true}	✓	✓	×	✓
ACBAB {A,A = false}	NA	NA	NA	NA
BAA {A,A = true}	×	×	×	✓
BAA {A,A = false}	NA	NA	NA	NA
ACAB {A = false, A = true}	✓	✓	✓	✓
ACBA {A = true, A = false}	✓	✓	×	✓
CA {C,A = true}	×	×	×	×
CA {C,A = false}	NA	NA	NA	NA

Table 4: Validation results of constraint Group 1

shows the traces that are present in the event log. In this column can also be seen whether the condition evaluates as *true* or *false*. The order in which the condition evaluation is listed corresponds to order in which the tasks are in the trace. For example, in trace `ACAB` {`A` = *false*, `A` = *true*} the first `A` in the trace the condition evaluates as *false*, whereas for the second `A` in the trace the condition evaluates as *true*. The additional columns correspond to the constraints tested, where a ✓ means the constraint is fulfilled and a × means the constraints is violated. NA states the trace was never activated. The test traces are chosen in such a way that for every constraint multiple traces are fulfilled, violated and not activated.

Group 2 contains constraints {*Precedence (P)*, *Alternate Precedence (AP)*, *Chain Precedence (CP)*}. The results of testing these constraints can be seen in Table 5.

Group 3 contains constraints {*Succession (S)*, *Alternate Succession (AS)*, *Chain Succession (CS)*, *Not Succession (NS)*, *Not Chain Succession (NCS)*}. The results of testing these constraints can be seen in Table 6.

Group 4 contains constraints {*Co-Existence (CE)*, *Not Co-Existence (NCE)*}. The results of testing these constraints can be seen in Table 7.

TRACE	P	AP	CP
AAB {B = true}	✓	✓	✓
AAB {B = false}	NA	NA	NA
BAA {B = true}	×	×	×
BAA {B = false}	NA	NA	NA
ABB {B,B = true}	✓	×	×
ABAB {B,B = true}	✓	✓	✓
ACBAB {B,B = true}	✓	✓	×
ACBAB {B,B = false}	NA	NA	NA
CBACB {B = true, B = false}	×	×	×
CABCB {B = false, B = true}	✓	×	×
ACBAB {B = false, B = true}	✓	✓	✓

Table 5: Validation results of constraint Group 2

All test results show the correct expected output, validating that the implementation functions correctly for every automaton. As for every constraint in an input Declare model a separate automaton is constructed, it can be concluded this implementation functions correctly for any Declare model.

5.4 CONCLUSION

This chapter presented an algorithm to allow for data-aware compliance checking of Declare models. The algorithm creates a final-state automaton for every relation constraint in a given data-aware Declare model. The algorithm determines for every final-state automaton and event log whether each log trace leads to a valid final-state. When a trace leads to a valid final-state, the constraint is fulfilled. When a trace does not lead to a valid final-state, the constraint is violated. Using this information the algorithm calculates a fulfillment ratio, the fraction of traces fulfilling the constraint, for every constraint in a given Declare model. To be able to assess the practical feasibility and relevance, the algorithm has been implemented in ProM in the form of two plug-ins.
The first plug-in, named "Create/Edit DeclareMap With Data", allows for a user to manually define variables. These variables are used to place guards on relation constraints on a given Declare model. The plug-in supports guards consisting of atoms of the form 'variable op constant' and 'variable op variable', where 'op' is a relational operator (e.g., =, <, or >). The constants can be strings, integers, doubles,

TRACE	S	AS	CS	NS	NCS
AAB {A,A,B = true}	✓	×	×	×	×
AAB {A,A,B = false}	NA	NA	NA	NA	NA
BA {B,A = true}	×	×	×	✓	✓
BAA {B,A,A = true}	×	×	×	✓	✓
BAA {B,A,A = false}	NA	NA	NA	NA	NA
ACDB {A,B = true}	✓	✓	×	×	✓
ACDB {A,B = false}	NA	NA	NA	NA	NA
ACADB {A,A,B = true}	✓	×	×	×	✓
ACBDB {A,B,B = true}	✓	×	×	×	✓
ABAB {A,B,A,B = true}	✓	✓	✓	×	×
BABA {B,A,B,A = false}	×	×	×	×	×
BCDB {B,B = true}	×	×	×	✓	✓
ACADB {A = true, A = true, B = false}	✓	×	×	×	✓
ACADB {A = false, A = false, B = true}	✓	✓	×	×	✓
ACADB {A = true, A = false, B = true}	✓	×	×	×	✓
ABDB {A= true, B = true, B = false}	✓	✓	✓	×	×
ACBDB {A = false, B = false, B = true}	✓	×	×	×	✓
ACBDB {A = true, B = false, B = true}	✓	×	×	×	✓

Table 6: Validation results of constraint Group 3

TRACE	CE	NCE
ACC {A,B = true}	×	✓
ACC {A,B = false}	NA	NA
ACB {A,B = true}	✓	×
ACB {A,B = false}	NA	NA
BCC {A,B = true}	×	✓
BCC {A,B = false}	NA	NA
BCA {A,B = true}	✓	×
ACB {A = true, B = false}	✓	×
ACB {A = false, B = true}	✓	×
ACC {A = false, B = true}	NA	NA

Table 7: Validation results of constraint Group 4

dates or booleans. Logical operators can be used to apply multiple conditions to a guard.

The second plug-in, "Declare Data-Aware Compliance Checker", takes a given Declare model and event log and applies the presented algorithm to determine the fulfillment ratio for every constraint in the given Declare model. The given Declare model can be enriched with guards, where the plug-in correctly checks whether the guards are satisfied or not. Relation constraints with no guards are treated as relation constraints with a guard that evaluates as true. The plug-in has been validated using multiple test event logs, showing the implementation functions as expected. Finally, the "Perform Predictions of Business Process Features" plug-in has been altered to allow for taking the "Declare Data-Aware Compliance Checker" results as input. Through this integration, it is now possible to correlate the violations of Declare constraints with other process characteristics, such as the occurrence of undesired events, the execution time or, even, the violations of other business rules.

The approach presented in [19] can be used to evaluate the conformance of a log with respect to a given Declare model. This approach determines whether a Declare constraint is violated or fulfilled and provides the user with an optimal alignment. Similarly to the presented data-aware compliance checking technique, this approach is based on the conversion of Declare constraints into automata and use these automata to identify violations and fulfillments. The approach focuses only on the control-flow perspective and as such the data perspective is not taken into account. The work presented in this thesis is therefore better suited for processes that contain decision points that

are governed by conditions, as it is able to evaluate data attributes.

The data-aware compliance checking technique presented in this thesis does not detect data violations. The data attribute values are read and in case the read data assignment evaluates the guard as true, the constraint is activated. In case a data attribute value is missing or incorrect, a constraint with a guard requiring this data assignment is never activated. As a result, it is possible a trace does not activate multiple constraints governed by guards due to a single incorrect data attribute value. The data-aware compliance checking technique presented in this thesis is not able to identify such data perspective violations.

In comparison, the approach suitable for a procedural approach presented in [8] separates control-flow, data and resource compliance checking to the possible extent, and provides integrated diagnostic information about both control-flow violations, and data and resource related compliance violations. However, this approach is based on the data-aware conformance checking technique presented in [16], a technique currently non-existent for a declarative approach.

6

DATA-AWARE COMPLIANCE CHECKING EVALUATION USING THE CASE STUDY

This chapter has been removed due to confidentiality.

7 CONCLUSION

This thesis provides both a case study done in cooperation with UWV as well as new contributions to allow for data-aware compliance checking of Declare models. The case study answers questions related to certain hypotheses and unclarities regarding the WIA Claim process. The WIA is an employment insurance in the Netherlands for clients that are still (partially) unfit for work after two years of illness. To conduct the case study, an event log is extracted containing event types that were selected based on multiple interviews with WIA domain experts at UWV. The event log additionally contains multiple variables on event level. To find answers to four business questions, which are formulated based on multiple interviews with WIA domain experts at UWV, multiple procedural (data-aware) process mining techniques are utilized in combination with a manually drawn procedural model of the WIA Claim process. The answers found for the business questions showed that certain hypothesises are indeed correct and that multiple unexpected process deviations are present. The drawn procedural model and extracted event log presented in this thesis in combination with the conducted procedural case study can serve as a basis for the WIA domain experts to conduct similar case studies in the future.

As UWV processes have a high variability both in terms of process execution and in terms of clients, a declarative approach might be more suited for a case study. This thesis presents a declarative control-flow miner and a data-aware compliance checking technique that can be used to conduct a similar case study but with a declarative approach.

This thesis presents the ProM implementation of a Declare miner based on the MINERful algorithm. This process discovery algorithm allows for the automated discovery of Declare models and is best suited for knowledge-intensive processes where the decisions taken over during the enactment of the process are usually fast and based on the expertise and intuition of the main actors. One of the biggest advantages is that this technique performs extremely well in terms of scalability. This implementation can serve as a way for business analysts to acquire a Declare model modelling the behavior of the business processes.

The data-aware compliance checking technique supports guards consisting of atoms of the form 'variable op constant' and 'variable op variable', where 'op' is a relational operator (e.g., =, <, or >). The

constants can be strings, integers, doubles, dates or booleans. Logical operators can be used to apply multiple guards to a constraint. Given a Declare model, possibly enriched with guards, and an event log the presented data-aware compliance checking algorithm determines for every constraint in the Declare model the set of traces fulfilling and the set of traces violating the constraint. Using these sets the fulfillment ratio, the fraction of traces fulfilling the constraint, is calculated for every constraint in a given Declare model. The technique correctly checks whether the guards that are specified on the relations are violated or not. Relation constraints with no guards are treated as relation constraints with a guard that evaluates as true. The implementation is validated using numerous test traces. The output presented by the data-aware compliance checking technique has been integrated in the correlation framework presented in [21]. Through this integration, it is now possible to correlate the violations of Declare constraints with other process characteristics, such as the occurrence of undesired events, the execution time or, even, the violations of other business rules.

The data-aware compliance checking technique is evaluated using the real-life event log extracted at UWV to assess the practical feasibility and relevance. This evaluation is done by repeating the compliance related part of the UWV case study using a declarative approach. The results found for the declarative approach are nearly similar to those found in the procedural approach. A deviation in the resulting statistics is present since the possible control-flow in the used Declare model is not entirely identical to the possible control-flow in the procedural model.

The main advantage found during the enactment of the declarative part of the case study, of a declarative approach over a procedural approach, is that the focus of compliance or conformance checking is on the relation between certain tasks when applying a declarative approach. In a declarative approach an analyst would not necessarily have to draw a model of the entire process to test the relation between two tasks. As a result an analyst could draw simple Declare models to test a certain relation between two (or more) tasks when an event log is available. This can be particularly useful when not every analyst has the same knowledge of the entire process.

The fact that similar results can be found to some of the existing procedural techniques, as well as the ability to quickly identify the relation between a subset of the tasks in the process, shows that the new data-aware compliance checking technique has a practical feasibility and relevance in a real-life case study.

7.1 ADVICE FOR UWV

This section has been removed due to confidentiality.

7.2 FUTURE WORK

This thesis has shown the practical feasibility and relevance of data-aware compliance checking of Declare models. The research in this thesis results in multiple opportunities for future work and research.

7.2.1 *Plug-in improvements*

One of the contributions presented in this thesis is the "MINERful Declare Miner" plug-in. In its current form, the plug-in presents four parameters that are user definable. The original MINERful implementation [7] contains additional options, such as excluding specific tasks from the miner or calculating some statistics. These options were not included, as they were irrelevant to the case study. These additional options could be implemented to give the "MINERful Declare Miner" a broader functionality and making it more suitable for case studies where these functionalities are desired. Additionally it might be desired to exclude specific constraint types from the mining process, this can be solved by giving the user the option to specify which Declare constraint types should be considered before the mining takes place.

The two plug-ins presented in this thesis that allow for data-aware compliance checking of Declare models are limited to processing relation constraints. It is possible to extend the plug-ins such that Declare models can enrich existence constraints with guards, namely by drawing final-state automatons mimicking the behavior of the constraint. The task to which the constraint is added should be considered as the *activation* in this case, such that these constraints can be checked for compliance. This functionality was not implemented in this thesis by choice. Enriching existence constraints with guards requires some additional research to determine how guards can best be used for these constraints. This is illustrated by the following example. A task can be executed multiple times with a condition that does not hold before it is executed with a condition that holds. As a result a specified upperbound in an existence template can easily be violated before a guard evaluates as true. For example, consider a task a with an existence template enriched with a guard stating it should be executed at most once if the blood pressure > 120. In case the trace of the person is $\langle$a[blood pressure = 110], a[blood pressure = 110], b[blood pressure = 120], a[blood pressure = 130]$\rangle$ the existence constraint would be violated even though the first two occurrences of a are not disallowed. With additional research, it might be possible to determine

how guards can best be used for these constraints. The data-aware compliance checking algorithm presented in Algorithm 2 is compatible with the extension of existence constraints, as the compliance checking functions similarly.

Recall that the data-aware compliance checking of a constraint is independent from the other constraints in the given Declare model. As such it is possible to improve the implementation to analyse the constraints in parallel, improving the performance

Additionally the two plug-ins that allow for data-aware compliance checking can be made more user friendly. The current implementation does not allow for mapping data attributes, i.e. the user has to specify data attributes in the guard that are identical to the data attribute names in the log. By implementing the possibility to map the data attributes in the Declare model with data attributes in the event log, the user does not receive error messages when for example a minor difference present in terms of lower or uppercase. Additionally the current implementation does not allow for exporting Declare models enriched with guards for later usage, which is desired when the model is frequently used.

7.2.2 *Data-Aware Conformance Checking of Declare Models*

The research presented in this thesis limits to data-aware compliance checking of Declare models, i.e. for each constraint is determined the set of traces violating and the set of traces fulfilling the constraint. The open issue for this approach is that no optimal alignment is presented and no fitness value is determined. In [16] a data-aware conformance checking technique is presented for a procedural approach by utilizing integer linear programming (ILP). This approach first relies on existing control-flow conformance checking techniques to build an alignment that only considers the control-flow perspective. Later a problem of ILP is constructed to obtain an optimal alignment that also takes the other process perspectives into account. For a procedural approach this approach works, as guards make the model more strict. As mentioned in this thesis, Declare works the other way around. Recall that placing a guard on a Declare constraint makes the constraint less strict. As a result first taking a control-flow alignment before constructing a problem of ILP does not work for a declarative constraint. Consider the example Declare model in Figure 23 and example event log $\sigma = AB$ where at all time Age = 40. An optimal control-flow alignment γ_1 found for the Declare model in Figure 23 and σ can be seen in Figure 24. Since data attributes are not considered, both task *C* and *D* should proceed task *B*. A conformance checking technique that would take data into account should find the optimal alignment γ_2, which can also be seen in Figure 24. The alignment pair $(\gg, C)$ in the control-flow alignment should never be present in the alignment that

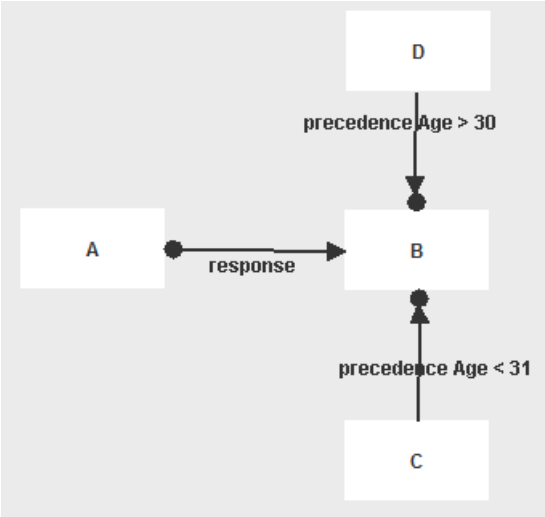

Figure 23: Example Declare model to illustrate a control-flow alignment versus an alignment taking data into account

$$\gamma_1 = \begin{array}{|c|c|c|c|} \hline A & \gg & \gg & B \\ \hline A & C & D & B \\ \hline \end{array} \qquad \gamma_2 = \begin{array}{|c|c|c|} \hline A & \gg & B \\ \hline A & D & B \\ \hline \end{array}$$

Figure 24: Alignments found for the Declare model in Figure 23 and example event log σ

takes data into account. For this reason a control-flow alignment can never be used as a basis for a declarative approach.

As a result a different approach should be used to perform data-aware conformance checking. A possible solution for this might be to utilize the automatons presented in this thesis. For each automaton that is not in a final-state the cheapest route, in terms of a cost function, to go to a final-state can result in an optimal alignment. Whether such an approach is feasible is left as future work.

The data-aware compliance checking technique presented in this thesis is not able to detect data violations. The data attribute values are read and in case the read data assignment evaluates the guard as true, the constraint is activated. In case a data attribute value is missing or incorrect, a constraint with a guard requiring this data assignment is never activated. As a result, it is possible a trace does not activate multiple constraints governed by guards due to a single incorrect data attribute value. The data-aware compliance checking technique presented in this thesis is not able to identify such data perspective violations. With data-aware conformance checking of Declare models, additional research can be done to diagnose such data perspective violations, similar to how the approach presented in [8] can do for a procedural approach.

A

CONFORMANCE CHECKING RESULTS

This appendix has been removed due to confidentiality.

B

BOTTLENECK ANALYSIS RESULTS

This appendix has been removed due to confidentiality.

C

CONSTRAINT AUTOMATONS

Appendix C contains the final-state automata that are defined for all relevant Declare relation constraints. Every automata considers a possible guard that is placed on the constraint.

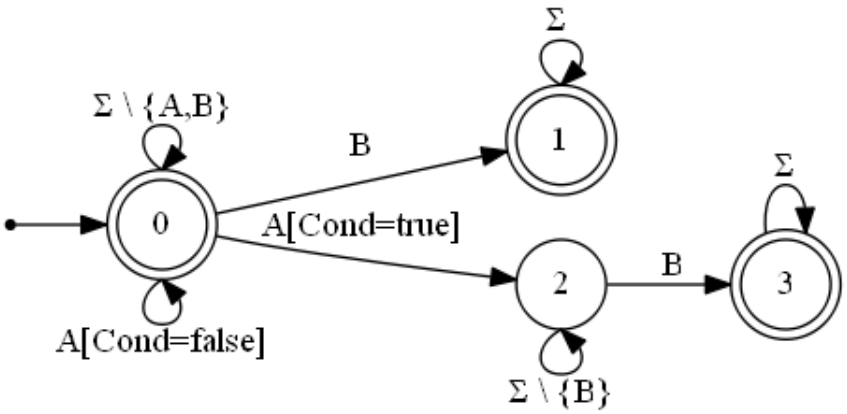

Figure 25: Responded Existence(A,B,Cond) - if A occurs and Cond holds, B must occur before or after A

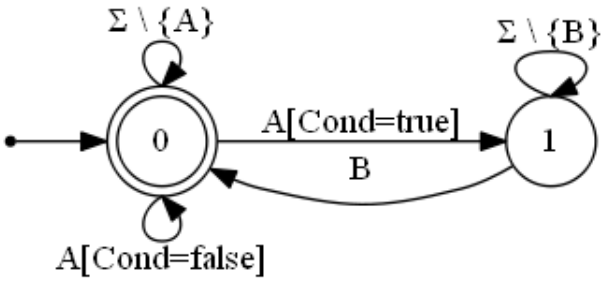

Figure 26: Response(A,B,Cond) - if A occurs and Cond holds, B must occur afterwards

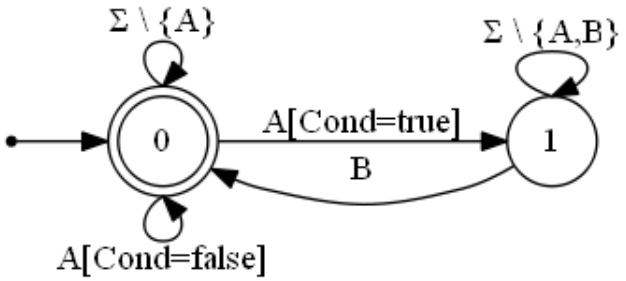

Figure 27: Alternate Response(A,B,Cond) - if A occurs and Cond holds, B must occur afterwards, without further As in between

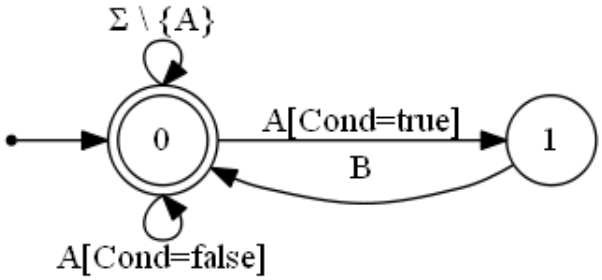

Figure 28: Chain Response(A,B,Cond) - if A occurs and Cond holds, B must occur next

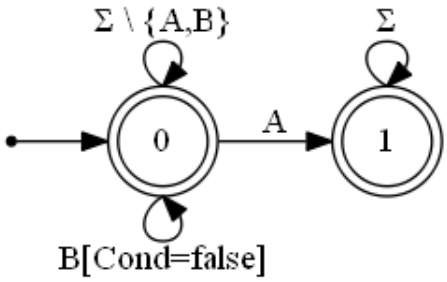

Figure 29: Precedence(A,B,Cond) - if B occurs and Cond holds, A must have occurred before

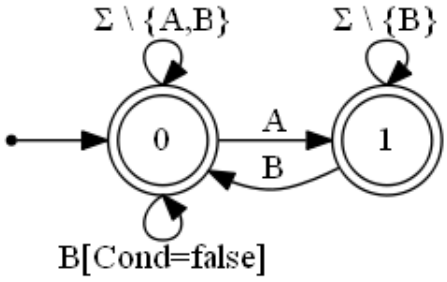

Figure 30: Alternate Precedence(A,B,Cond) - if B occurs and Cond holds, A must have occurred before, without other Bs in between

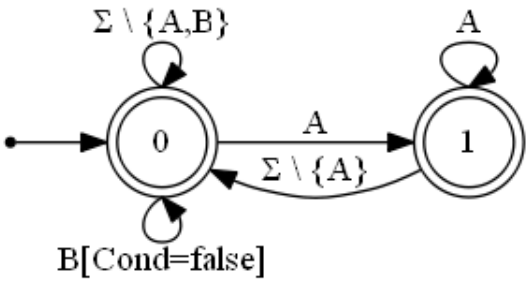

Figure 31: Chain Precedence(A,B,Cond) - if B occurs and Cond holds, A must have occurred immediately before

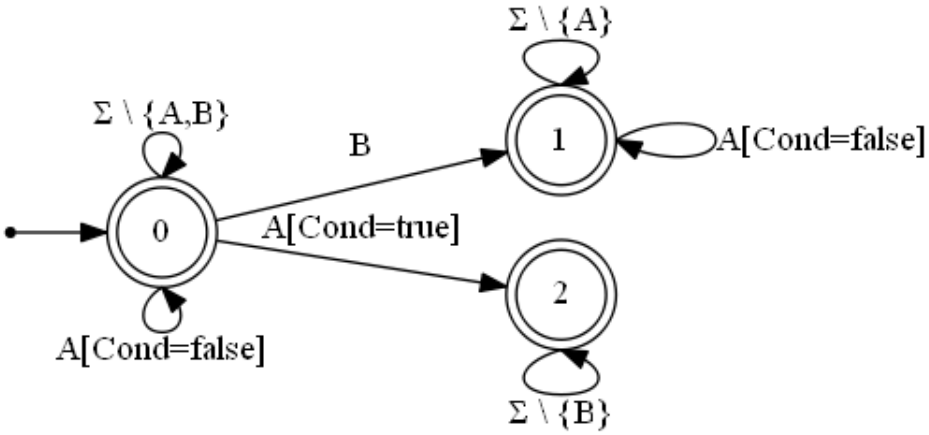

Figure 32: Not Responded Existence(A,B,Cond) - if A occurs and Cond holds, B can never occur

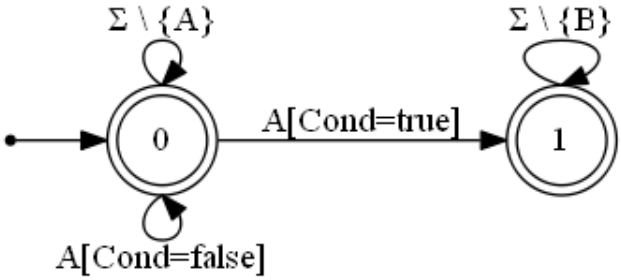

Figure 33: Not Response(A,B,Cond) - if A occurs and Cond holds, B cannot occur afterwards

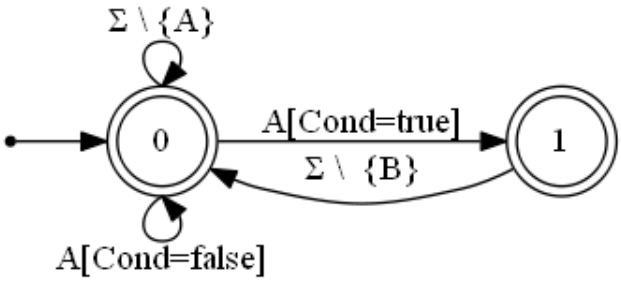

Figure 34: Not Chain Response(A,B,Cond) - if A occurs and Cond holds, B cannot be executed next

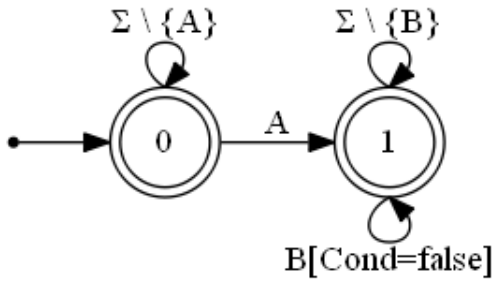

Figure 35: Not Precedence(A,B,Cond) - if B occurs and Cond holds, A cannot have occurred before

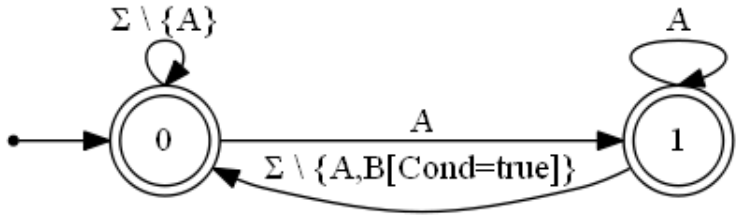

Figure 36: Not Chain Precedence(A,B,Cond) - if B occurs and Cond holds, A cannot have occurred immediately before

D

COMPLIANCE CHECKING RESULTS

This appendix has been removed due to confidentiality.

BIBLIOGRAPHY

[1] A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance Checking using Cost-Based Fitness Analysis. *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference*, 2011.

[2] A. Rozinat, and W.M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Information Systems*, 33:64–95, 2008.

[3] C. Di Ciccio, A. Marrella, and A. Russo. Knowledge-intensive processes: An overview of contemporary approaches. *Proceedings of the 1st International Workshop on Knowledgeintensive Business Processes*, 861:33–47, 2012.

[4] C. di Ciccio, and M. Mecella. On the Discovery of Declarative Control Flows for Artful Processes. *ACM Trans. Manage. Inf. Syst.*, 5(4):1–37, January 2015.

[5] C. Di Ciccio, M.H.M. Schouten, M. de Leoni, J. Mendling. Declarative Process Discovery with MINERful in ProM. *Proceedings of the BPM Demo Session 2015*, 1418:60–64, 2015.

[6] C. Hill, R. Yates, C. Jones, and S.L. Kogan. Beyond predictable workflows: Enhancing productivity in artful business processes. *Syst. J.*, 4(45):663–682, 2006.

[7] C. di Ciccio. MINERful implementation, 2015. URL `http://github.com/cdc08x/MINERful`.

[8] E. Ramezani, V. Gromov, D. Fahland, and W.M.P. van der Aalst. Compliance Checking of Data-Aware and Resource-Aware Compliance Requirements. *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, 2014.

[9] F. Mannhardt, M. de Leoni, H.A. Reijers, and W.M.P. van der Aalst. Balanced Multi-Perspective Checking of Process Conformance. *Computing*, 2015.

[10] F.M. Maggi. Declarative Process Mining with the Declare Component of ProM. ceur-ws.org, 2013.

[11] F.M. Maggi, M. Dumas, L. Garcia-Banuelos, and M. Montali. Discovering Data-Aware Declarative Process Models from Event Logs. *Business Process Managament*, 2013.

[12] H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM6. *Proceedings of Information Systems Evolutions*, 2011. URL `http://www.processmining.org/tools/prom`.

[13] J. Desel, and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.

[14] M. de Leoni, and F. Mannhardt. Road Traffic Fine Management Process, 2015.

[15] M. de Leoni, and W.M.P. van der Aalst. Data-Aware Process Mining: Discovering Decisions in Processes Using Alignments. *Proc. of the 28th ACM symposium on Applied Computing*, 2013.

[16] M. de Leoni, and W.M.P. van der Aalst. Aligning Event Logs and Process Models for Multi-Perspective Conformance Checking: An Approach Based on Integer Linear Programming. *Proc. of the 10th International Conference on Business Process Management*, 2013.

[17] M. de Leoni and W.M.P. van der Aalst. The FeaturePrediction Package in ProM: Correlating Business Process Characteristics. *CEUR Workshop Proceedings*, 1295:26–30, 2014.

[18] M. de Leoni, F.M. Maggi, and W.M.P. van der Aalst. Aligning Event Logs and Declarative Process Models for Conformance Checking. *Business Process Management*, 7481:82–97, 2012.

[19] M. de Leoni, F.M. Maggi, and W.M.P. van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Information Systems*, 2014.

[20] M. de Leoni, J. Munoz-Gama, J. Carmona, and W.M.P. van der Aalst. Decomposing Conformance Checking on Petri net With Data. *Proc. of 22nd International Conference on Cooperative Information Systems*, 2014.

[21] M. de Leoni, W.M.P. van der Aalst, and M. Dees. A General Framework for Correlating Business Process Characteristics. *Business Process Management*, 8659:250–266, 2014.

[22] M. Pesic, and W. M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. *BPM Workshops*, 2006.

[23] Eindhoven University of Technology. ProM Nightly build, 2015. URL `http://www.promtools.org/prom6/nightly`.

[24] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers. IEEE Task Force on Process Mining: Process Mining Manifesto. *BPM 2011 Workshops, LNBIP*, 99:169–194, 2011.

[25] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers. Imperative versus Declarative Process Modelling Languages: An Empirical Investigation. *BPM Workshops, LNBIP*, 2011.

[26] Process Mining Group, Math&CS department. Performance Analysis with Petri Net, 2009. URL `http://www.processmining.org/online/performanceanalysiswithpetrinet`.

[27] W. M. P. van der Aalst, and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. *WS_FM*, 2006.

[28] M. Westergaard. Better algorithms for analyzing and enacting declarative workflow languages using ltl. *Proceedings of the 9th Business Process Management Conference*, 6896:83–98, 2011.

[29] W.M.P. van der Aalst. *Business Information Systems: A Process-Oriented Approach*. Eindhoven University of Technology, Eindhoven, Noord-Brabant, Eindhoven, 2007.

[30] W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Eindhoven, Noord-Brabant, Netherlands, 2011.

[31] W.M.P. van der Aalst, A. Adriansyah, and B.F. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

[32] W.M.P. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - R & D*, 2009.