UNIVERSITY OF TARTU

Institute of Computer Science

Software Engineering Curriculum

CLIVE TINASHE MAWOKO

Aligning Data-Aware Declarative Process Models and Event Logs

Master's Thesis (30 ECTS)

Supervisor(s):

Prof Fabrizio M. Maggi

Tartu 2019

Acknowledgements

Aligning Data-Aware Declarative Process Models and Event Logs

Abstract

Conformance checking, a branch of process mining, allows analysts to determine whether the execution of a business process matches the modeled behavior. Process models can be procedural or declarative. Procedural models dictate the exact behavior that is allowed to execute a specific process whilst declarative models implicitly specify allowed behavior with the rules that must be followed during execution. The execution of a business process is represented by event logs. Conformance checking approaches check various perspectives of a process execution including control-flow, data and resources. Approaches that checks not only the control-flow perspective, but also data and resources are called multi-perspective or data-aware approaches. The approaches provide more deviation information than control-flow based techniques. Alignment based techniques of conformance checking have proved to be advantageous in both control-flow based and data-aware approaches. While there exist several data-aware approaches for procedural process models that are based on the principle of finding alignments, there is none so far for declarative process models.

In this thesis, we adapt an existing technique for finding alignments of logs and data-aware procedural models to declarative models. We implemented our approach as a plugin of the process mining framework ProM and evaluated it using event logs with different characteristics.

Andmeteadlike deklaratiivsete mudelite ja sündmuste logide joondamine

Abstrakt

Vastavusanalüüs on haru protsessikaevanduses, mis võimaldab analüütikutel saada aru, kas äriprotsesside sooritused järgivad mudeldatud käitumist. Protsesside mudelid võivad olla nii protseduurilised kui ka deklaratiivsed. Kui protseduurilised mudelid kirjeldavad ära täpsed võimalikud tegevused, siis deklaratiivsed mudelid kirjeldavad reeglid, mis peavad olema protsessi sooritusel olema järgitud. Äriprotsesside täitmiste hoiustamiseks kasutatakse sündmuste logisid. Vastavusanalüüsi meetodid kontrollivad erinevaid protsessi sooritusega seotud vaateid, milleks on juhtimisvoog, andmed ja ressursid. Meetodid, mis käsitlevad endas lisaks juhtimisvoole ka andmeid ning ressursse kutsutakse mitmevaatelisteks või andmeteadlikeks lähenemisteks. Mitmevaatelised meetodid annavad rohkem informatsiooni kõrvalekallete kohta võrreldes juhtimisvoogudel põhinevate meetoditega. Joondustel põhinevad vastavusanalüüsi meetodid on olnud edukad nii juhtimisvool põhinevate kui ka andmeteadlike lähenemiste puhul. On olemas mitmeid joondamisel põhinevaid andmeteadlikke lähenemisi protseduuriliste mudelite jaoks, kuid deklaratiivsete mudelite jaoks need puuduvad.

Antud töös on kohandatud olemasolev meetod, mis võimaldab sooritada vastavusanalüüsi andmeteadlike protseduuriliste mudelite puhul, kasutades logide joondustel põhinevat meetodit, võimaldamaks kasutamist ka deklaratiivsetel mudelitel. Deklaratiivsetel mudelitel rakendatav meetod implementeeriti moodulina protsessikaeve keskkonna ProM jaoks ja hinnati implementatsiooni kasutades erinevaid sündmuste logisid.

Märksõnad: **Protsessikaevandus, Deklaratiivsed protsessimudelid, Andmeteadlik vastavusanalüüs, Joondamine**

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Table of Contents

List of Figures

List of Tables

1 Introduction

Business process execution in companies are usually supported by process-aware information systems which store an event log containing every activity/event that goes through the system. Conformance checking is a branch of process mining that verifies whether the recorded behavior in an event log matches the modeled behavior in a process model [1]. This type of analysis is critical in domains such as process auditing, security and risk analysis.

Business process models can either be procedural or declarative. In procedural models, the finishing of one activity may enable the execution of other activities. They dictate the exact behavior that is allowed to execute a specific process and examples include Petri nets and BPMN. Procedural models are ideal for processes that are predictable. Declarative models implicitly specify the allowed behaviors with rules that must be followed during execution [2]. Also known as constraint-based models, in declarative models, everything is allowed unless it is specified as forbidden. An example is Declare [3]. Declarative models are not restrictive like procedural models but allows every behavior except those that are explicitly listed as constraints. Declarative models are best suited for processes that are dynamic where users can use their discretion to choose which path to follow e.g. in healthcare systems.

Conformance checking can either be control-flow based or multi-perspective. Control-flow based approaches only consider the ordering of activities, ignoring other perspectives such as data, time and resource perspectives. As the name suggests, multi-perspective approaches, also known as data-aware approaches, consider the control-flow plus data, time and resource perspectives. While control-flow based approaches can discover deviations, more deviations can be discovered by introducing the data, timing and resource aspects [4]. [1] also reiterates this, giving examples of deviations related to activities that are executed by a wrong resource; such deviations cannot be discovered using control-flow based approaches only because the order of events remains correct until details of the resource that executed the activity are checked. Four quality dimensions can be used in conformance checking. The most well-known is fitness which states that a log with high fitness contains only behavior that is in line with the model. Other dimensions are precision, simplicity and generalization. Precision describes the degree to which a model allows unlikely behavior given the observed behavior in an event log [5]. According to [6] simplicity shows how

the *resulting process model is readily understandable* and generalization *refers to the ability of a process model to abstract from the behavior that is documented in the logs*

The focus of this thesis is on data-aware or multi-perspective conformance checking of declarative models. The majority of the data-aware conformance checking approaches for procedural models are based on the principle of alignment. An alignment can show how an event log can be changed to perfectly fit a process model. [4] [7] [8] [9] [10] [11] use the principle of alignment in their approaches. The main advantage realized is that of easily providing diagnostics. Alignments show exactly where the deviations are, and their severity. [8] points out that a log-model alignment can be used as input of a variety of other techniques such as techniques for cleaning an event log by removing traces that should not be used in further analysis. Another use case is conformance checking of an event log against a process model highlighting exactly where deviations occur. Also, alignments can be used to repair process models based on the behavior present the log. However, to the best of our knowledge, a data-aware conformance checking approach for declarative models based on the principle of alignment does not exist in the literature. This has motivated us to come up with **an alignment-based data-aware conformance checking approach for declarative models.** The technique takes as input, a declarative model and an event log, and outputs for each trace in the event log, an alignment that shows how the trace can be replayed on the model. A similar approach was presented in [8] but only considered the control-flow perspective. We extend this approach to also consider data variables. We use the A* algorithm to compute the alignments like in [7]. We implemented our approach as a plugin of the open source framework ProM [12].

This thesis is structured as follows:
- Chapter 2 gives a background of the related tools and techniques used in this research.
- Chapter 3 gives the details of our conformance checking approach.
- Chapter 4 describes how the solution was implemented and presents the results of the evaluation.
- Chapter 5 discusses the related work as presented in the literature and we conclude in Chapter 6.

2 Background of Study

This chapter discusses the background elements required to understand the rest of this thesis. These include event logs, declarative process modeling, finite state automata, integer linear programming and the A* algorithm.

2.1 Process mining and event logs

Process mining is the extraction and/or analysis of knowledge from process execution data aimed at discovering, monitoring and improving processes. Process execution data is also known as event logs. Three branches of process mining include *automatic process discovery, performance analysis and conformance checking* [6]. Process execution data, in the form of event logs, contain information about process instances as a collection of traces. One trace corresponds to one process instance. Each trace consists of a sequence of events, which are ordered according to their time of execution. An event refers to a well-defined step in a business process, known as an activity. Traces and events contain attributes. As key-pair values, attributes are used to store additional information such as name of trace/event, data elements associated with an event, the executor of an event, timestamps etc.

For uniformity, the *IEEE Task Force on Process Mining promotes the usage of the eXtensible Event Stream (XES) format* [6]. XES is an XML-based standard for event logs. It is aimed at providing a generally-acknowledged format of how event logs are stored, exchanged and analyzed [13]. Figure 1 shows an example trace with events and attributes in XES format. The complete metamodel of the XES standard is represented in Figure 2.

```xml
<trace>
    <string key="concept:name" value="Case No. 3"/>
    <event>
        <string key="concept:name" value="A"/>
        <string key="lifecycle:transition" value="complete"/>
        <date key="time:timestamp" value="2019-01-04T21:23:14.691+01:00"/>
        <int key="x" value="5"/>
        <string key="y" value="Sam"/>
    </event>
    <event>
        <string key="concept:name" value="B"/>
        <string key="lifecycle:transition" value="complete"/>
        <date key="time:timestamp" value="2019-01-04T21:30:11.691+01:00"/>
        <int key="x" value="5"/>
        <string key="y" value="Philip"/>
    </event>
</trace>
```

Figure 1: An example trace with events and attributes in XES format
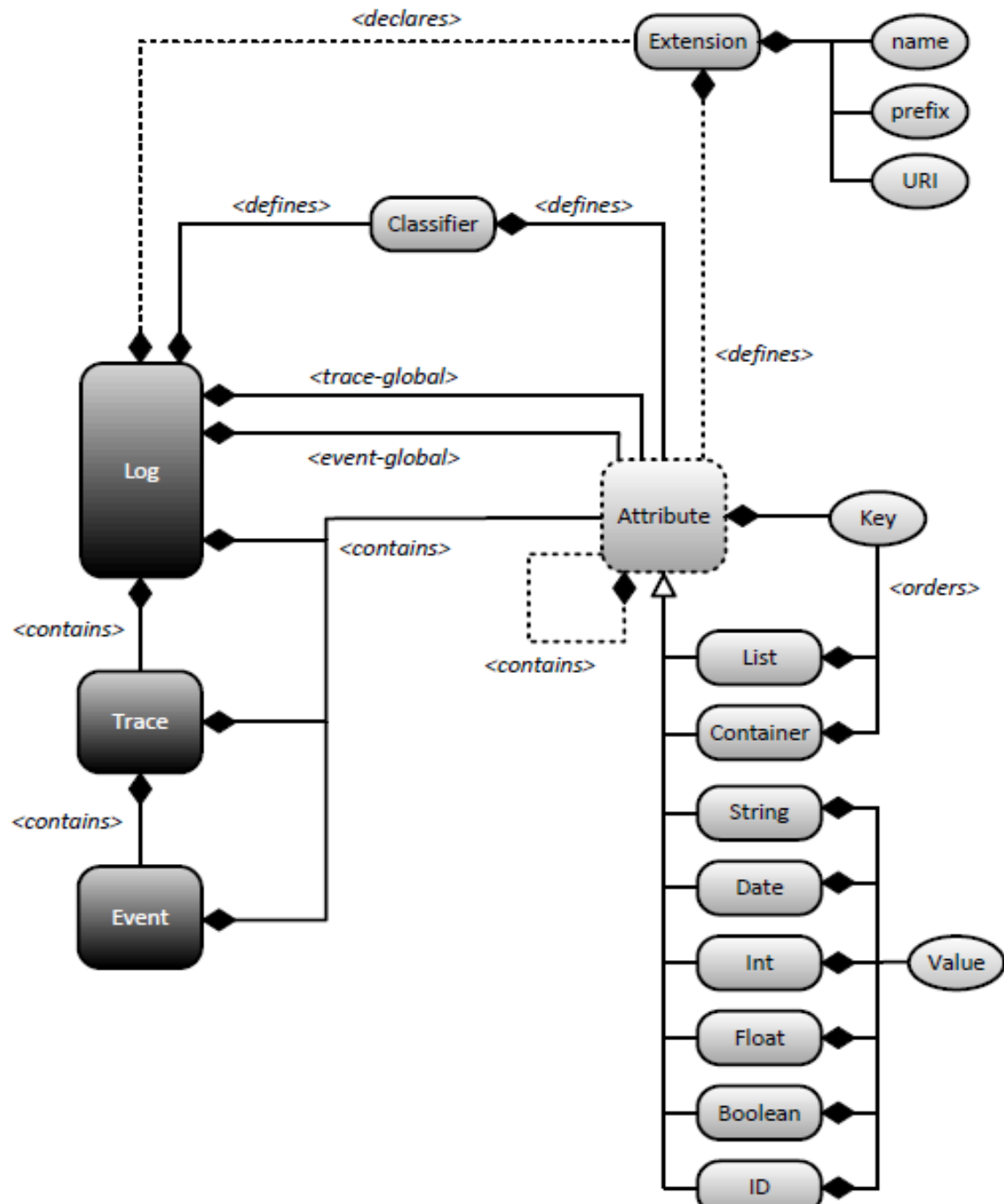
Figure 2: A complete meta-model UML 2.0 class diagram for the XES standard *[24]*

2.2 Declarative modeling

Declarative models implicitly specify allowed behaviors with rules that must be followed during execution [2]. In this thesis, we focus on a declarative language called Declare, which was

introduced in [3]. Declare is a constraint-based process modeling language. A Declare model consists of a set of activities and a set of constraints defined on the activities. Any activity can be executed in a Declare model if it does not violate any of the set constraints. The example in Figure 3, taken from [8], shows how a process can be modeled using the Declare language. The process is executed a travel agency that handles health insurance claims. Activities are shown as rectangles and constraints are the connectors between the activities.
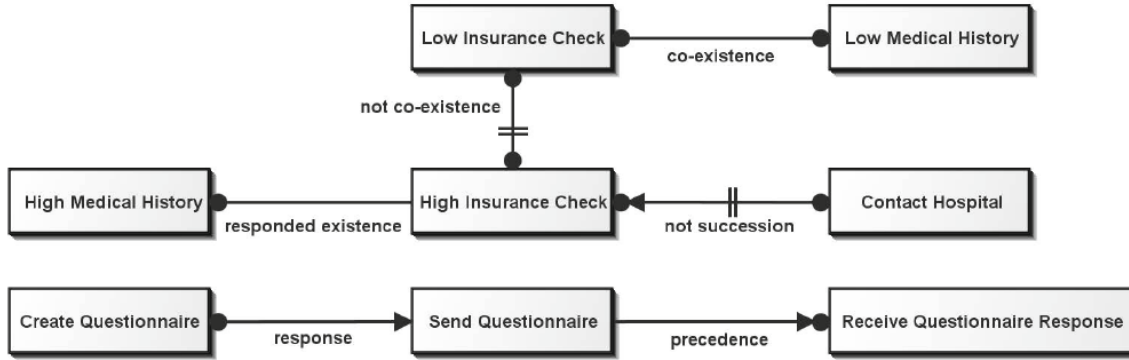
Figure 3: Declare model with 8 activities and 6 constraints *[8]*

The model in Figure 3 indicates that *Low Insurance Check and High Insurance Check cannot coexist in the same process instance as depicted by the not co-existence constraint. The co-existence constraint indicates that Low Insurance Check and Low Medical History always occur together in any order. High Medical History can only be executed together with High Insurance Check, in any order. This is what the responded existence constraint entails. The not succession constraint depicts that Contact Hospital cannot be followed by High Insurance Check in the same process instance. The response constraint entails that Create Questionnaire is followed eventually by Send Questionnaire. The questionnaire can be filled and received (Receive Questionnaire Response) if it was sent before in the precedence constraint.*

The Declare language specifies a set of standard templates that are used in creating constraints. That is, *constraints are concrete instantiations of templates* [14]. The use of templates makes the model comprehension independent of its formal implementation. This approach helps analysts to work with the graphical representation without the knowledge of the underlying formulas.

2.2.1 Declare templates

Declare templates can be divided into four major groups: existence, relation, negative relation and choice.

1. Existence templates

This is a set of unary templates. Unary templates are only applicable to a single activity. Table 1 shows the list of existence templates including init, end, atmostone, participation and absence.

Template Name	Description	Graphical Representation
Init(A)	The process instance must start with activity A	INIT a
End(A)	The process instance must end with activity A	END a
Participation(A)	Activity A must be executed at least once in a process instance	1..* a
AtMostOne(A)	Activity A must not be executed more than once in a process instance	0..1 a
Absence(A)	Activity A should not be executed in a process instance	0 a

Table 1: List of Declare existence templates

2. Relation templates
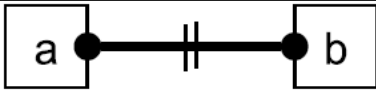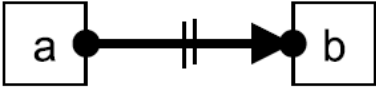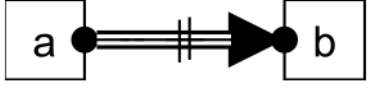
These are rules affecting two activities and their relationship. The occurrence of one determines the occurrence of the other. They are either ordered or unordered. In ordered relation templates, the activities should occur in a specifies sequence while in unordered templates, activities can occur in any order. Table 2 shows a list of the relation templates.

Template Name	Description	Graphical Representation
Responded existence(A,B)	If A occurs at least once, B must occur at least once either before or after A	
Co-existence(A,B)	If either A or B occurs, the other activity has to occur as well	
Response(A,B)	If A occurs, B must eventually occur	
Precedence(A,B)	A must occur before B	
Alternate response(A,B)	A stricter version of response which enforces that no other A should occur between the execution of A and B eventually following.	
Alternate precedence(A,B)	A stricter version of precedence which enforces that no other B must occur between B being preceded by A (e.g. ABB is not allowed).	
Chain response(A,B)	If A occurs, then B must occur immediately after A	
Chain precedence(A,B)	If B occurs, then A must have occurred immediately before B	

Table 2: List of Declare relation templates

3. Negative Relation templates
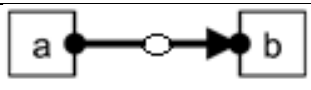
This group describes the negated versions of the relation templates. Like relation templates, they can also be ordered or unordered. Table 3 shows the list of negative relation templates.

Template Name	Description	Graphical Representation
Not RespondedExistence(A,B) *Not Co-Existence(A,B)*	Either A or B can be executed but not both	
Not response(A,B) *Not precedence(A,B)* *Not succession(A,B)*	Before the execution of B there cannot be A. After the execution of A, B cannot occur	
Not chain response(A,B) *Not chain precedence(A,B)* *Not chain succession(A,B)*	A and B should never follow each other directly	

Table 3: List of Declare negative relation templates

4. Choice templates

In choice templates, one must choose to execute an activity from a set of given activities. Table 4 below shows a list of the choice templates.

Template Name	Description	Graphical Representation
Choice(A,B)	At least A or B must be executed	
Exclusive Choice(A,B)	At least A or B must occur but not both	

Table 4: List of Declare choice templates

The above lists of Declare constraint templates are not exhaustive. Further details can be found in [15].

2.2.2 Declare with data

As stated in section 2.1, an activity is associated with attributes that store additional information related to the execution of that activity. This information can include the data variables written by the activity, the time event was executed and the resource that executed that activity etc. This information is also known as the *payload* of an activity. The Declare constraints discussed so far only focus on the ordering and execution of activities (control-flow) ignoring the data aspects. To include data, the Declare constraints are extended by adding three conditions on the payload of the

activity namely, activation, correlation and temporal condition. These are added in the following format:

Constraint_name(activity_A, activity_B)[activation condition][correlation condition][temporal condition]

- The activation condition specifies a condition on data that must hold true when the constraint is activated. This means that the constraint should not be activated when this condition is false. This condition is only on data of the activation payload.

- The correlation condition is related to the occurrence of the target activity. Target is said to have occurred only if this condition is fulfilled. This condition can include data from both activation and target payloads.

- The temporal condition is used to specify time distances between activities.

The technique presented in this thesis is only focused on constraints with activation conditions.

To specify the payload in a constraint, the dot operator can be used. In particular, we indicate with A.x, the data attribute x attached to activity A. The above example can be written as follows:

Co-existence(LowInsuranceCheck, HighInsuranceCheck)[LowInsuranceCheck.amount<300][][]

NB: The other condition brackets are empty because they are not being considered.

2.3 Finite State Automata

A finite state automaton (FSA) is a mathematical model of computation based on a labelled transition system. The transition system can be defined as $A = (\Sigma, \Psi, \psi_0, \delta, F)$ where

- *Σ is the finite input alphabet*
- *Ψ is a finite, non-empty set of states*
- *$\psi_0 \in \Psi$ is an initial state*
- *$\delta \in \Psi \times \Sigma \to \Psi$ is the state transition function*
- *$F \subseteq \Psi$ is the non-empty set of final or accepting states ($F \neq \emptyset$)*

A state transition function is a function, such that, given a state and a character (input), returns a new state, the target state (if defined). The FSA can be used to evaluate a set of inputs, one at a time, using the state transition functions. *A finite path π of length n over A is a sequence $\pi = \langle \pi^1, \dots, \pi^n \rangle$ of tuples $\pi^i = \langle s^{i-1}, a^i, s^i \rangle \in \delta$, where a is an input character, for which the following condition hold true:*

i. π^1 , the first tuple, is such that $s^0 = \psi_0$ (it starts from the initial state), and

ii.   the starting state of π^i is the target state of π^{i-1} : $\pi = \langle (s^0, a^1, s^1)(s^1, a^2, s^2), \dots , (\ ^{sn-1}, a^n, s^n) \rangle$

A sequence of characters of length n is said to be accepted by the automaton A if $\pi^n = \langle s^{n-1}, a^n, s^n \rangle$ s.t $s^n \in F$ [14]

2.4 Integer Linear Programming

Integer linear programming (ILP) is an approach for achieving optimization in a mathematical model given a linear objective function and a set of linear constraints. ILP solves the *problem of either maximizing or minimizing a linear function* with respect to given conditions or constraints [16]. The goal is to find optimal variable values that solve the given ILP problem. It is called an integer problem when all the variables are integers. When some but not all are none integer, such as strings, it is called a mixed integer problem. Consider the example in Table 5 taken from [17]. The aim is to find a real number x, given certain conditions. The initial step defines the minimum and maximum possible values of x. This means that x is a number between m and M, i.e., $m < x < M$. The first condition states that x should be less than 10. This changes the solution for x to be $m < x < 10$. The next condition states that x should be more than 5. This changes the lower limit for x and the solution becomes $5 < x < 10$. If we continue adding conditions to x, we reduce to range of possible values for x thereby moving closer to the actual value. It is important to note, however, that not all new conditions necessarily change the range for x. All previous conditions must remain true. That is, if a new condition intends to change either the lower limit or upper limit, the range only changes if the previous values remain true otherwise it remains the same or the value is lost (no solution is found). An example is the condition in step 4 which states that the value of x should be less than 100. In this case, an upper limit of 100 violates a previous conditions stating an upper limit of 10. However, since setting the upper limit to 10 fulfils both conditions, the range remains $5 < x < 10$. Other conditions can fix the value to a constant. By adding condition 5, $x = 9$, we set both the minimum and maximum at 9. If a new condition violates any of the previous conditions, it means a solution cannot be found. An example is the condition in step 6 which changes the upper value of x to 8 when a previous condition set it to 9. Both conditions cannot coexist and hence there will be no solution.

Step	Condition	Range	Has Solution
0	init	m < x < M	true
1	x < 10	m < x < 10	true
2	x > 0	0 < x < 10	true
3	x > 5	5 < x < 10	true
4	x < 100	5 < x < 10	true
5	x = 9	x = 9	true
6	x < 8	*no solutions*	false

Table 5: *An example ILP Problem [17]*

2.5 A* Algorithm

A* algorithm is a search algorithm intended to find the path with the lowest overall cost between two nodes in a direct graph with costs associated to nodes. Given a graph, V, and a node $v_0 \in V$ as the start node, A* explores adjacent nodes until reaching any node of the given target set. There is a cost associated with every node v determined by the evaluation function $f(v) = g(v) + h(v)$ where $g(v)$ is a function that returns the smallest path from v_0 to v and $h(v)$ is a heuristic function that estimates the path cost from v to its preferred target node. Function $h(v)$ should underestimate the distance of a path from one node to its preferred target node. If so, A* is guaranteed to find a path with the lowest overall cost. The algorithm keeps a priority queue of nodes to be visited and high priority is given to nodes with the lowest costs so as to visit those first. The algorithm works iteratively; at each step, a node v with the highest priority is drawn from the queue. If it belongs to the target set, then the algorithm terminates returning that node. Otherwise, v is expanded and every successor is added to the priority queue with a cost $f(v')$.

3 Conformance Checking Framework

This section discusses the main focus of this thesis, i.e. the framework for computing alignments between a data-aware declare model and an event log. We illustrate the framework using examples about how different scenarios can be handled. In general, the approach takes as input an event log and a data-aware Declare model. It then produces for each trace in the event log, an alignment that best describes how the trace can be replayed on the model without violating any constraint. The alignment shows how the trace conforms to the model and to quantify conformance, we calculate fitness.

3.1 Event logs

Let A_L be a set of log activities and X_L be the set of log variables defined over a universe U of values. An event is pair $e = \langle a, V \rangle$ where $a \in A_L$ is the activity to which e refers and $V : X_L \nrightarrow U$ is a function that associates variables $x \in X$ to a value $V(x)$. Denote with $E = (A_L \times (X_L \nrightarrow U)$ the set of possible events. An event log $\mathcal{L}$ is a multiset of traces, where each trace is a sequence of events in E.

Example 1: *Let us assume a log with the following trace:*

$\sigma = \langle (B, \{x=3;y="Sam"\}), (A, \{x=5;y="Philip"\}), (C, \{x=5\}), (D, \{x=1;y="Philip"\}) \rangle$

3.2 Data-aware Declare Models

A data-aware Declare model consists of a set of Declare constraints each of which can be represented through a final-state automaton with conditions attached to transitions. Declare constraints are defined over a set A of activities and a set X of variables. Without any loss of generality, we assume that $A \subseteq A_L$ and $X \subseteq X_L$ where, as discussed above, A_L and X_L indicate the set of log activities and variables, respectively.

Potential deviations of an event log from a reference data-aware Declare model can be identified by a mapping between events in the log and execution traces admissible by the model. Declare allows for the execution of any activity, even those that are not in the model. The set of activities in the event log but not specified by the process model, denoted $A_L \setminus A$, do not need to be distinguished for conformance checking. This enables us to reduce the space of the allowed

behaviors. However, we cannot completely abstract from such activities because some constraints use LTL's next operator (e.g., the chain response and chain precedence constraints). Therefore, in the remainder, any activity in $A_L \setminus A$ is mapped onto the special tick activity $\checkmark$.

Example 1 (cont): *Assuming we have a Declare model with the following constraints*

  Response(A,B)[A.x>3][][]

  Absence(D)[D.x>3&&D.y=="Sam"][][]

Because only A, B and D are specified in the model, activity C is converted into $\checkmark$. The above log trace, σ, is converted to $\bar{\sigma}$ as follows:

  $\bar{\sigma} = \langle(B, \{x=3;y="Sam"\}), (A, \{x=5;y="Philip"\}), (\checkmark, \{x=5\}), (D, \{x=1;y="Philip"\})\rangle$

We now formally introduce the concepts of data-aware Declare constraints and models. A Data-Aware Declare model is a set of declare constraints along with the definition of the set of variables, the potential values taken on by those variables and the definition of writing operations that indicate the set of variables which the different activities are prescribed to assign/update the value of.

Definition 1 (Data-Aware Declare model). *A data-aware Declare model D = (A; X; U; Val; I; Write; Π) consists of:*

- *a set A of activities;*
- *set X of variable names (process data);*
- *a (potentially infinite) set U of variable values;*
- *a function I : X → U that assigns the variable's initial values*
- *a function Val : X → 2^U defining the admissible values for each variable x∈X, i.e., val(x) is the (potentially infinite) domain of variable x;*
- *a function Write : A → 2^X that define the variables that are written by each activity a ∈ A.*
- *a set Π of data-aware declare constraints over A and V*

We extend the finite state automata discussed in section 2 to include the concept of guards. A guard is a data condition that is assigned to a transition, such that, the transition will only fire if the condition is true.

Definition 2 (Data-aware constraint automaton). *Let A be a set of activities and let X be a set of variables. Let Guard(X) be the set of all possible guards defined over the set X of variables. The*

constraint automaton $\mathcal{A}_{A;X} = (\Sigma, \Psi, \psi_0, \delta, G, F)$ *over a set A of activities and a set X of variables is a final-state automaton which accepts precisely those traces that satisfy a Declare constraint, where:*

- $\Sigma = A \cup \{\checkmark\}$ *is the input alphabet;*
- Ψ *is a finite, non-empty set of states;*
- ψ_0 *is an initial state;*
- $\delta \in \Psi \times \Sigma \to \Psi$ *is the state-transition function*
- $G \in \Psi \times \Sigma \to Guard(X)$ *is a guard function that assigns a guard to a transition.*
- $F \subseteq \Psi$ *is the set of final states.*

Example 1 (cont): *The response constraint mentioned above can be represented as follows:*

- $\Sigma = \{A;B;D;\checkmark\}$
- $\Psi = \{S0;S1\}$
- $\psi_0 = \{S0\}$
- $\delta = \{ S0 \times \{A\} \to S1; S0 \times \{\Sigma \setminus A\} \to S0; S1 \times \{\Sigma \setminus B\} \to S1; S1 \times \{B\} \to S0; \}$
- $G = \{ (S0 \times \{A\} \to S1) \leftarrow (x>3), (S0 \times \{A\} \to S0) \leftarrow !(x>3)\}$
- $F = \{ S0\}$

The absence constraint can be represented as follows:

- $\Sigma = \{\checkmark;\checkmark;\checkmark;D\}$
- $\Psi = \{S0\}$
- $\psi_0 = \{S0\}$
- $\delta = \{ S0 \times \{\Sigma \setminus D\} \to S0\}$
- $G = \{ (S0 \times \{D\} \to S0) \leftarrow !((x>3)\&\&(y=="Sam"))\}$
- $F = \{ S0\}$

The constraint automata can be depicted using finite-state machines as shown in Figure 4

(i) Response constraint automaton

(ii) Absence constraint automaton
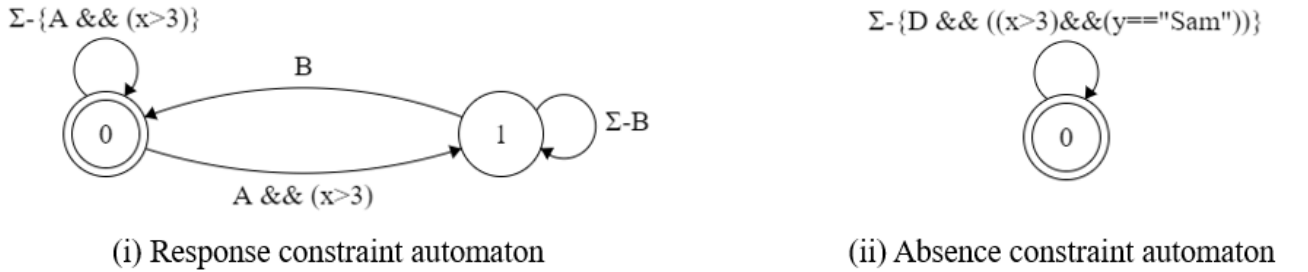
Figure 4: Example Declare constraint automata

In order to define the replay semantics of a declare model, it is necessary to introduce the concept of state of a data-aware process that is modelled through a data-aware Declare model, hereafter shortened as state of a data-aware Declare model:

Definition 3 (State of a data-aware Declare model). *The state of a data-aware Declare model D = (A; X; U; Val; I; Write; Π) is a pair (V; S) consisting of*
- *the (current) assignments V : X → U of values to variables;*
- *the current state S of each automaton in Π, namely, for each constraint automaton $\mathcal{A}_{A;X}$ = (Σ_i, Ψ_i, ψ_{0i}, δ_i, G_i, F_i) ∈ Π, $S(\mathcal{A}_{A,X})$ ∈ Ψ_i.*

The following definition introduces the concept of legitimate execution of steps, activities or ticks, of a data-aware Declare model with certain write operations:

Definition 4 (Legal step executions). *Let (V; S) be the state of a data-aware Declare model D = (A;X;U; Val; I; Write; Π). It is possible to execute an activity or a tick t ∈ A ∪{√} with a set of write operations as defined in a function v: X →U iff*

- *The domain of v coincides with Write(t);*

- *For each x ∈ Write(t), v(x) ∈ Val(x);*

- *For each A_i = (Σ_i, Ψ_i, ψ_{0i}, δ_i, G_i, F_i) ∈ Π, $\delta_i(A_i; t)$ is defined, $G_i(A_i; t)$ holds wrt. variable assignment in V*

- *This yields to a new state (V'; S') such that, for all A_i = (Σ_i, Ψ_i, ψ_{0i}, δ_i, G_i, F_i) ∈ Π, $S'(A_i)$ = $\delta_i(A_i; t)$. Function V' is constructed from V as follows: for each x ∈ X, if x ∈ Write(t) V'(x) = v(x), otherwise V'(x) = V(x). This is denoted as (V; S) $\xrightarrow{(t,v)}$ (V'; S').*

Definition 5 (Sets of execution traces of a data-aware model). *Let D = (A; X; U; Val; I; Write; Π) be a data-aware Declare model. Let (V0; S0) be the initial state, namely V0 = I and, for all A_i = (Σ_i, Ψ_i, ψ_{0i}, δ_i, G_i, F_i) ∈ Π, $S_0(A_i)$ = ψ_{0i} . A trace σ = ⟨(t_1; v_1); … ; (t_n; v_n)⟩ is an execution trace of D iff (V0; S0) $\xrightarrow{(t_1,v_1)}$ (V1; S1) $\xrightarrow{(t_2,v_2)}$ … $\xrightarrow{(t_n,v_n)}$ (Vn; Sn) and, for all Ai = (Σ_i, Ψ_i, ψ_{0i}, δ_i, G_i, F_i) ∈ Π, $S_n(A_i)$ ∈ F_i*

3.3 Data-aware Alignment of Declare Models

To find an alignment between an event log to a model, moves in the log are related to moves in the model. Some moves in the log may not be mimicked by the model resulting in a 'no move' in

the model. Other moves in the model cannot be reproduced in the log resulting in a 'no move' in the log. We explicitly denote such "no moves" by $\gg$.

Definition 6 (Alignments). *Let $D = (A,X,U, Val, I, Write, \Pi)$ be a data-aware Declare model. Let $\tilde{A}_X = A \cup \{\checkmark\}$. Let $M_D = (\tilde{A} \times (X \to U)) \cup \{\gg\}$. An alignment move is represented by a pair $(S_L, S_M) \in (M_D \times M_D) \setminus \{(\gg,\gg)\}$ such that:*

- *(S_L, S_M) is a move in log if $S_L \in (M_D \setminus \{\gg\})$ and $S_M = \gg$,*
- *(S_L, S_M) is a move in model if $S_M \in (M_D \setminus \{\gg\})$ and $S_L = \gg$,*
- *(S_L, S_M) is a correct synchronous move if $S_L = S_M$.*
- *(S_L, S_M) is an incorrect synchronous move if $S_L, S_M \in (M_D \setminus \{\gg\})$ and, denoted $S_L = (a_L, v_L)$ and $S_M = (a_M, v_M)$, $a_L = a_M$ and $v_L \neq v_M$.*

A complete alignment between D and a log trace $\sigma \in (\tilde{A} \times (X \to U))^*$ is a sequence of legal alignment moves $\langle (s^1_L; s^1_M); ...; (s^n_L; s^n_M) \rangle$ such that, ignoring every $\gg$, $\langle s^1_L; ... ; s^n_L \rangle$ is equal to σ, and $\langle s^1_M; ... ; s^n_M \rangle$ is an execution trace of D.

Example 1 (cont): *Given a Declare model with the constraints depicted in Figure 1, and a log trace $\bar{\sigma} = \langle (B, \{x=3;y="Sam"\}), (A, \{x=5;y="Philip"\}), (\checkmark, \{x=5\}), (D, \{x=1;y="Philip"\}) \rangle$. The following are valid complete alignments:*

$\gamma_{1=}$

S_L	B{x=3;y="Sam"}	A{x=5;y="Philip"}	$\gg$	$\checkmark${x=5}	D{x=1;y="Philip"}
S_M	B{x=3;y="Sam"}	A{x=5;y="Philip"}	B{}	$\checkmark${x=5}	D{x=1;y="Philip"}

$\gamma_{2=}$

S_L	B{x=3;y="Sam"}	A{x=5;y="Philip"}	$\checkmark${x=5}	D{x=1;y="Philip"}
S_M	B{x=3;y="Sam"}	A{x=3;y="Philip"}	$\checkmark${x=5}	D{x=1;y="Philip"}

Note that the aim is to find an alignment with the least deviation cost. This is called an optimal alignment. A cost function on legal moves is first introduced, and then generalized to alignments in order to define the severity of a deviation. The definition below was taken from [7].

Definition 7 (Cost Function & Optimal Alignment). *Let $D = (A, X, U, Val, I, Write, \Pi)$ be a data-aware Declare model. Let $\sigma \in (\tilde{A} \times (X \to U))^*$ be a log trace. Let $\tilde{A}_X = A \cup \{\checkmark\}$. Let M_D be*

the set of all legal alignment moves. Let κ be a cost function that assigns a non-negative cost value to each legal move: κ: M_D → $\mathbb{R}_0^+$. The cost of an alignment γ between σ and D is the sum of the costs of all constituent moves: $K(γ) = Σ_{m∈γ}κ(m)$. Alignment γ is an optimal alignment if, for any alignment γ' of D and σ, $K(γ) ≤ K(γ')$.

The process domain and the specific process model determines the cost of each legal move. The cost function $κ$ needs to be defined for each specific setting because it can be used to influence one type of explanation of deviations over the others. Note that an optimal alignment does not need to be unique, i.e. multiple complete alignments with the same minimal cost may exist.

Example 1 (cont): *Assuming a cost function such that $(S_L, \gg) = (\gg, S_M) = 10$ and incorrect synchronous move attracts a penalty of 1 for each mismatching variable. No cost is assigned to a correct synchronous move. The complete alignments, $γ_1$ and $γ_2$ would have the following costs: $K(γ_1)= 10$, $K(γ_2)= 1$. Therefore according to the cost function, $γ_2$ is the optimal alignment because it has the least cost.*

In the example above, the cost for an incorrect synchronous move is assigned per each mismatching variable. This means that, if an activity writes 2 variables and the alignment process assigns different values for both variables from the ones observed in the log trace, a penalty is assigned for each variable individually. If the cost is 1 for each variable, then the cost of this move would be 2.

3.4 A* Algorithm

The process of finding an optimal alignment of a log trace σ and a Data-aware Declare model *D* can be complicated. Especially when dealing with declarative models, the search space can be very large despite the introduction of the ✓ events. The A* algorithm provides a solution to finding the least expensive path in a directed graph with nodes and edges. We adopt the use of the A* algorithm the same way it is adopted in [7] but adapting it to declarative models. Just like in [7], an opportune search space needs to be defined. Each node of the search space is associated to a different alignment which is a prefix of some complete alignment between σ and D. As a directed graph, the edges connecting the nodes are weighted based on the predefined cost structure.

Instead of building the search space at once, the search space is built incrementally. Starting with the source node $\gamma_0 = \langle \rangle$, an empty alignment, successors are obtained by adding one move to it until the target node is reached. The target node is a set of all the complete alignments of σ and D. An alignment is complete when the log projection (i.e. log activities excluding $\gg$ activities) is equal to the initial trace and the process projection is a valid prefix of the process model (i.e. all automata in their final state). Each successor/node, γ, in the search space is associated with a cost based on the evaluation function $f(\gamma) = g(\gamma) + h(\gamma)$. $g(\gamma)$ is the cost of the alignment from γ_0 and is obtained using the following function:

$$g(\gamma) = K^{min}.|\gamma| + K(\gamma) \ where,$$

- K^{min} *is the smallest value of cost K, added to guarantee termination.*
- $|\gamma|$ *is the size of the current alignment's log projection, that is counting the number of log activities considered so far* excluding $\gg$
- $K(\gamma)$ *is the cost of the alignment according to the predefined cost structure.*

All moves leading to an alignment, contribute to be final cost of that alignment. This means that the value of $g(\gamma)$ strictly increases as moves are added to alignment prefixes. That is $g(\gamma'') > g(\gamma')$ *since* $g(\gamma'') = g(\gamma') + K^{min}.|\gamma''| + K(\gamma'')$.

$h(\gamma)$ estimates the path cost from γ to the target node (a complete alignment) and is denoted by the following function

$$h(\gamma) = K^{min}.(|\sigma|-|\gamma|), \ where,$$

- $|\sigma|$ *is the size of the log trace*
- $|\gamma|$ *is the size of the current alignment's log projection*

A* algorithm works with a priority queue, Q, with the node with the least cost at the top of the queue. Since we build our graph incrementally, a node is picked from the top of the queue. Initially, this is an empty alignment, γ_0. Control flow successors, *ctrl_succ$_{\sigma,D}$(γ)*, between the log trace σ and D are sought. These are found by adding exactly one legal move, (S_L,S_M), to the current prefix based on the current position in the log trace and the states of the automata. The move only considers the control flow perspective, ignoring all the write operations. Control flow successors cannot make proper search space nodes since they do not include the data perspective.

Data operations need to be added to each control flow successor, γ_C, before it is committed to the queue. This is the augmentation stage, *augment(γ_C),* as described by in Algorithm 1. Data values need to be chosen such that the costs are kept at a minimum and no guard is violated. To this aim we use Integer Linear Programming (ILP) to solve the problem of data assignments. The data values returned by the ILP solver is the data to be used in the alignments. If no solution is found by the ILP solver, as denoted by Γ, the successor is discarded otherwise the alignment cost, $f(\gamma)$, is computed and the alignment is added to the queue.

After the augmentation of all the successors, a new node is picked from the top of the queue for a new round of the A* algorithm. If the node corresponds to a complete alignment, then the search stops, and optimal alignment is found. If it is not a complete alignment, control flow successors are also sought and augmented, then added to the queue where necessary. The process continues until a complete alignment is picked from the queue. A* algorithm guarantees that the first complete alignment found is optimal.

Algorithm 1

Input: Data-aware Declare Model D = (A, X, U, Val, I, Write, Π), a log trace $\sigma = \langle e_0, \ldots,$ $e_n \rangle$ and a cost structure K
Result: An optimal Alignment γ
$\gamma \leftarrow \gamma_0 = \langle \, \rangle$
Cost-ordered queue $Q \leftarrow \langle \, \rangle$
foreach $\mathcal{A}_{A,X} = (\Sigma, \Psi, \psi_0, \delta, G, F) \in \Pi$ **do**
    $S0(\mathcal{A}_{A,X}) \leftarrow (\Sigma, \Psi, \psi_0, \delta, G, F)$
end
while logProjection(γ) $\neq \sigma \wedge \forall \mathcal{A}_{A,X} = (\Sigma, \Psi, \psi_0, \delta, G, F) \in \Pi. \, S(\mathcal{A}_{A,X}) \notin F$ **do**
    foreach γ'_C in *ctrl_succ$_{\sigma,D}$ (γ)* **do**
        $\gamma' \leftarrow$ *augment(γ_C)*
        if $\gamma' \neq \Gamma$ **then**
            $f(\gamma') \leftarrow g(\gamma') + h(\gamma')$
            addToQueue($Q, \gamma', f(\gamma')$)
        end
    end
    $\gamma \leftarrow$ pickAndRemoveLowestCost (Q)
end

To illustrate how the algorithm works, we finish off example 1 with the graph of the search for the optimal alignment shown in Figure 6. Each circle is a node representing a prefix of some complete

alignment and is labeled with the cost of that alignment. The edges are labeled with the move in the form of (S_L, S_M). The shaded nodes are the nodes picked from the top of the queue at each iteration and they are numbered in the order they are picked. The node that represent the optimal alignment is shown by a double line. To simplify the graph, the variable assignments are not shown.

Example 1 (cont): *Figure 5 shows how Multiple Integer Linear Programming (MILP) is employed to assign values to one of the control flow successors. Figure 5a shows a control flow successor, that is, with no write operations. In order to assign data values to the specified variables, each variable is given a placeholder with a number showing the ith time the variable is being written as shown in 5b e.g. x_1, y_1, x_2 and y_2. The values need to be chosen such that no guard is violated. Also, the aim is to minimize the deviation between the log trace values and the process values. The placeholder variables become the MILP variables. Two sets of constraints can be observed in figure 5c. The first set involves the guards associated with the model against the MILP variables. The second set of constraints involves a boolean variable, for each MILP variable, that shows whether the MILP variable is assigned the same value as the one observed in the log trace. For instance, $\hat{x}_1$ is given the value 0 iff x_1 is given the same value as the log trace, that is, $x_1=3 \Leftrightarrow \hat{x}_1=0$. The objective function is the total cost of deviations as determined by the sum of the boolean variables.*

Log Trace	Process
B {x=3, y="Sam"}	B {}
A {x=3, y="Philip"}	A {}

(a) *Control flow successor*

Log Trace	Process
B {x=3, y="Sam"}	B {x=x_1, y=y_1}
A {x=3, y="Philip"}	A {x=x_2, y=y_2}

(b) *The skeleton of all augmentations of the control-flow successor*

$$\min \hat{x}_1 + \hat{y}_1 + \hat{x}_2 + \hat{y}_2$$
$$x_2 > 3$$
$$x_1 = 3 \Leftrightarrow \hat{x}_1 = 0$$
$$y_1 = Sam \Leftrightarrow \hat{y}_1 = 0$$
$$x_2 = 3 \Leftrightarrow \hat{x}_2 = 0$$
$$y_2 = Philip \Leftrightarrow \hat{y}_2 = 0$$

(c) *The MILP problem to find the augmentation with the lowest cost of deviations*

Figure 5: Augmentation of a control-flow successor of an alignment prefix
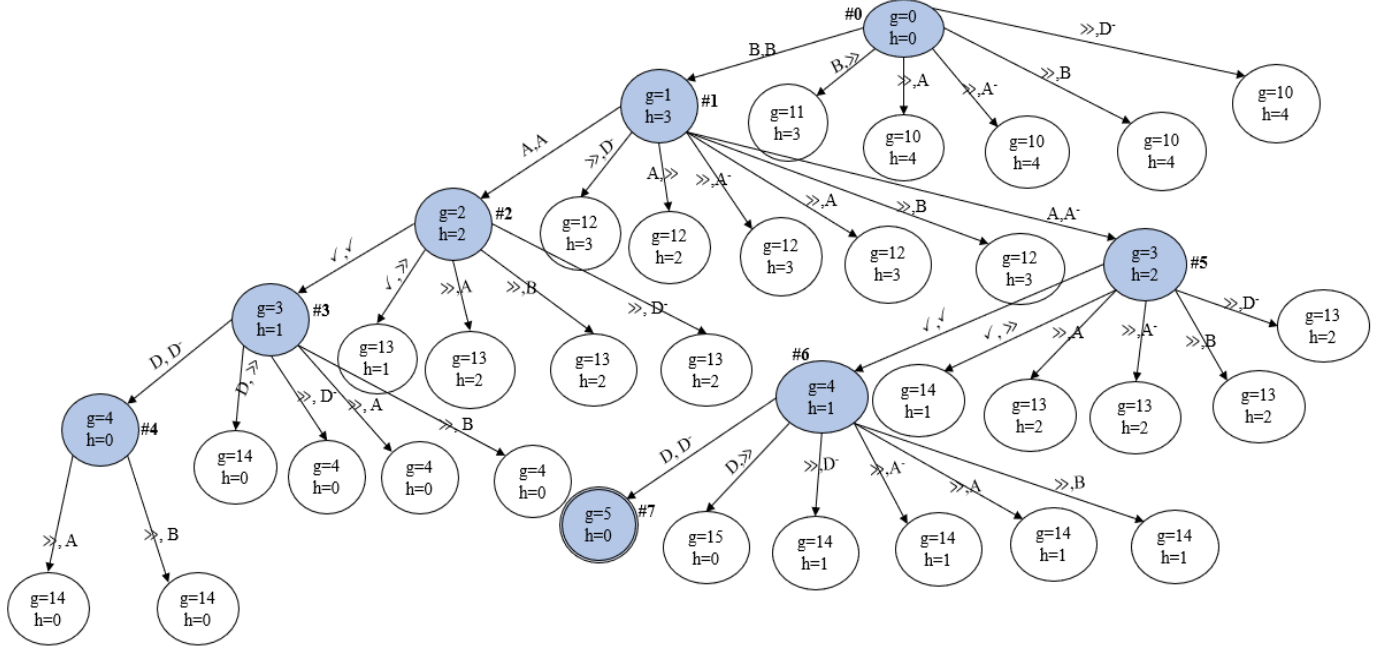
Figure 6: A* graph for example 1

Please note, the activities A⁻ and D⁻ represent the negated versions of the activation activity. That is, the activities associated with the negative guard in the automata. To understand the figures above, the augmentation of the control-flow successor illustrated in Figure 5 is only for the node #2 successor in Figure 6. Node #7 in the Figure 6 graph represents the path with the optimal alignment i.e. the path with the nodes #0, #1, #5, #6 and #7.

3.5 Search Space Reduction

The A* algorithm search space for declarative models can be too large causing the algorithm to perform poorly in terms of speed. This is because declarative models are more flexible and allows more behavior than procedural models. In order to improve on performance, many of the search space nodes can be pruned because they are equivalent to one another.

Definition 8 (Alignment Equivalence). *Let $D = (A, X, U, Val, I, Write, \Pi)$ be a data-aware Declare model. Let $\sigma_L \in (\tilde{A} \times (X \rightarrow U))^*$ be a log trace. Let $\tilde{A}_X = A \cup \{\checkmark\}$. Let $\mathcal{A}_\pi = (\Sigma_\pi, \Psi_\pi, \psi_{0\pi}, \delta_\pi, G_\pi, F)$ be the constraint automaton for $\pi \in \Pi$. Let γ' and γ'' be alignments of σ'_L and σ'_M, and of σ''_L and σ''_M where σ'_L and σ''_L are prefixes of σ_L and σ'_M and σ''_M are prefixes of model traces*

21

in P_D. Let $\psi'_\pi = \delta^*_\pi(\psi_{0\pi}, \sigma'_M)$ and $\psi''_\pi = \delta^*_\pi(\psi_{0\pi}, \sigma''_M)$ be the states reached by $\mathcal{A}_\pi$ when replaying σ'_M and σ''_M on it. Alignments γ' and γ'' are equivalent with respect to D, if $\sigma'_L = \sigma''_L$ and, for all $\pi \in \Pi$, $\psi'\pi = \psi''\pi$. We denote this with $\gamma' \sim_D \gamma''$.

If two partial alignments γ' and γ'' are equivalent, they can be extended by the same sequence of moves. Also, the least expensive path to the target node, h(γ') and h(γ''), from the two alignments is the same. It is however necessary to only visit one of them, i.e. the one with the lowest g cost.

3.6 Degree of Conformance

In order to calculate the degree of conformance, we limit this thesis to the calculation of fitness for each trace. Fitness is calculated the same way as described in [8]. The cost of the optimal alignment is divided by the cost of the reference alignment. The reference alignment is the alignment with only moves in the model and moves in the log as follows:

$\gamma^{\text{ref}}_{(\sigma_L, \sigma_M)} =$

L	σ_1^L	...	σ_n^L	≪	≪	≪
M	≪	≪	≪	σ_1^M	σ_1^M	σ_1^M

We use the reference alignment because it has the maximum cost possible. The following definition was taken from [8].

Definition 9 (Fitness). *Let D = (A, X, U, Val, I, Write, Π) be a data-aware Declare model. Let σ_L be a log trace. Let $\gamma \in \Sigma_A^*$ be an optimal alignment of σ_L and D. Let σ_M $\gamma\#_M$ be the aligned model trace. Let $\gamma^{\text{ref}}_{(\sigma_L, \sigma_M)} = \Sigma_A^*$ be the reference alignment of σ_L and D. The fitness score of σ_L with respect to D is defined as follows:*

$$Fitness(\sigma_L, D) = 1 - \frac{K(\gamma_{\sigma_L})}{K\left(\gamma^{ref}_{(\sigma_L, \sigma_M)}\right)}$$

Therefore $Fitness(\sigma_L, D) = 1$ if the optimal alignment only have moves in both, i.e. no deviations. The fitness is 0 if the optimal alignment is equal to the reference alignment. Please note, the fitness returned is always a positive fraction, that is a value between 0 and 1.

Example 1 (cont): *Using the cost function, such that $(S_L, \gg) = (\gg, S_M) = 10$, incorrect synchronous move attracts a penalty of 1 for each mismatching variable and 0 penalty for a correct synchronous move. The optimal alignment, γ_1 below has a cost of 1, whereas the cost of reference alignment, γ_{ref}, is 60. Therefore, the fitness = 1-1/60 = 0.983.*

$\gamma_1 =$

S_L	B{x=3;y="Sam"}	A{x=5;y="Philip"}	✓{x=5}	D{x=1;y="Philip"}
S_M	B{x=3;y="Sam"}	A{x=3;y="Philip"}	✓{x=5}	D{x=1;y="Philip"}

$\gamma_{ref} =$

S_L	B	A	✓	D	≪	≪
S_M	≪	≪	≪	≪	A	B

4 Implementation and Evaluation

This section details how the solution was implemented and evaluation.

4.1 Implementation

Two ProM plugins are used in the implementation of our solution. ProM is a Java based open source framework that provides a platform for developers to easily develop and/or extend process mining algorithms. We implemented the conformance checking framework, as described in chapter 3, in a ProM plugin called Data Aware Declare Replayer. The plugin takes as input a Data-Aware Declare model and an event log and outputs alignments, for each log trace. For visualization, we implemented another plugin called Data Aware Alignment Result Visualizer. It takes the output of our Data Aware Declare Replayer plugin as input and provides a clean way to visualize alignments for each trace as well as showing the fitness and related statistics. Figure 7 shows a screenshot of the details of trace alignments in the Data Aware Alignment Result Visualizer. The left panel shows the list of traces labelled with the trace name as well as its fitness. Upon clicking one of the traces, the middle part is filled with the details of that trace showing the following information:

- The top part shows quick statistics such as the fitness of the trace, number of moves in both log and model (correct synchronous moves), number of moves in both log and model but with different data (incorrect synchronous moves), number of moves in log only and number of moves in model only.

- Each event is represented by a colored rectangular box labeled with event name or "TICK". Each color represent the move type of each event i.e. green represents a correct synchronous move, white represents an incorrect synchronous move, yellow represents a log move and pink represents a model move.

- On hover on each event, more details is displayed showing the data values associated with the event.

- Traces can also be sorted by trace names and there is also a filtering functionality by trace name.

Figure 7: Screenshot of a single trace alignment details

Since this was an extension of the work presented in [8], a lot of code from the Declare Replayer[1] plugin implementation was reused in the implementation of our Data Aware Declare Replayer plugin. As mentioned earlier, the LP Solver library [18] was used for solving ILP problems. We deal with strings by mapping them to integers.

The source code of our plugins together with direction of use can be downloaded from https://github.com/Clyvv/DataAwareDeclareReplayer.

4.2 Evaluation

We provide 2 ways of evaluating our solution. Firstly, we look at the correctness of the results of our solution. Then we look at solution feasibility in terms of performance. In all the experiments, the Data Aware Declare models were created and or edited using ProM plugins, Simple Declare Designer and Simple Declare Editor. The generation of synthetic logs was necessitated by a tool called MP-Declare Log Generator described in [19].

---

[1] https://svn.win.tue.nl/repos/prom/Packages/DeclareChecker/

4.2.1 Solution verification
Verification of a single Data Aware Declare constraint with a simple condition

We start with a simple example with only one constraint and a simple logical condition as shown in Figure 8. This was tested against the synthetic event log trace shown in Table 6.
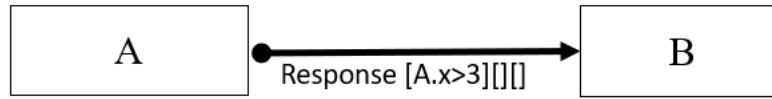

Figure 8: Single constraint declare model with a simple condition

Activity	B	A	C	D
Data variables	x == 3	x == 5	x == 5	x == 1
	y == "Sam"	y == "Philip"		y == "Philip"

Table 6: Single event log trace

The trace shows that event A activates the constraint because the value of x is greater than 3. If that happens, it means the trace is in violation of the response constraint because event B is not eventually followed by a B. The result is of course influenced by the cost function. If the cost function assigns more cost for control flow deviations over data writing costs the result might change. The results from our solution is in Figure 9.

Figure 9: Alignment result of a Declare model with a simple condition

26

Figure 9a shows the result where the cost function assigns more cost to data deviation whilst Figure 9b is showing the result where the cost assigned to control flow deviation is more that data deviation cost. In Figure 10b, the x value of event A is changed from 5 to 3 and hence the constraint is not activated. Events C and D are shown as TICK events because as discussed earlier, events not specified in the model but appearing in the log trace are represented in replaying as tick events to reduce the search space of the A* algorithm. Both results are correct according to their respective cost function.

Verification of a Data Aware Declare constraint with a string condition

This test will demonstrate that our solution can handle conditions that require string values. We use the same event log trace in Table 6 and the following Data Aware Declare model.
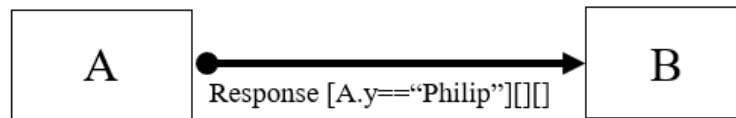

Figure 10: A single constraint Declare model with a string condition

The condition states that A must be followed by B if the value of y is equal to Philip. That is the constraint is only activated if A is executed with value of y == "Philip".


Figure 11: Alignment result of a model containing a string condition

27

2 different optimal alignments were output by our solution. Figure 11a was run with a cost function that assigns more cost to deviations associated with data writing costs. The response condition is activated because the value of y is equal to Philip. But because there is no execution of event B after A, the constraint is violated and hence the introduction of a move in model of event B. The result shown in 11b was run using a cost function that assigns more cost value to control flow deviations. In this case, it is less costly to change the value of the y variable and not activate the constraint. Hence, the incorrect synchronous move on activity A. Note that ILP managed to assign a value to y that is not equal to Philip.

Verification of a Data Aware declare constraint with multiple conditions

In this test we show how our solution deals with constraints with multiple conditions that is, conditions combined by && and/or ||. To demonstrate, we use the Data Aware Declare model in Figure 12, and the event log trace in Table 6.
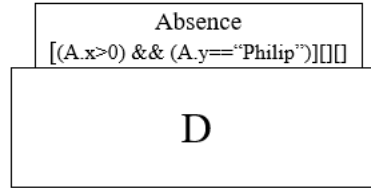

Figure 12: A single constraint declare model with multiple conditions

The model shows an Absence condition that dictates that event D must not be executed if x > 0 and y == "Philip". The log trace shows that event D has values x == 1 and y == "Philip". The values fulfils the given condition and hence D cannot be executed and hence the trace is in violation.

Figure 13: Alignment results of a Declare model with constraint with multiple conditions

Our solution rightfully detected this misconformances and provided the 2 optimal alignments in Figure 13. Figure 13a was run with a cost function that assigns more cost value to the deviations associated with write operations while in Figure 13b the cost function assigned more cost value to control flow deviations. In Figure 13a, a move in log is introduced to avoid activating/violating the constraint. In 13b, an incorrect asynchronous move is introduced because it is less costly to change 1 variable value than removing the whole activity hence the value of Philip was changed.

Verification of multiple Data Aware Declare constraints

In this section we see whether our solution can correctly align more complex Declare models, i.e. with multiple constraints of different types. We use the same event log trace in Table 6 and the data aware Declare model depicted in Figure 14.
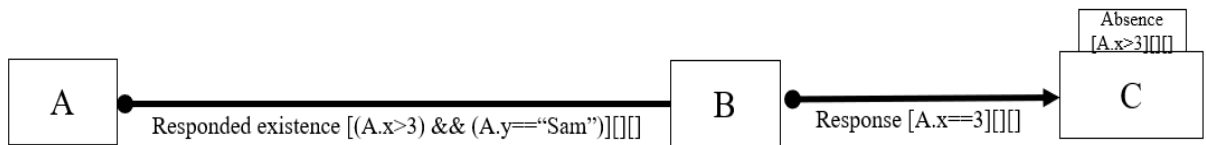

Figure 14: A Data Aware Declare Model with 3 constraints

The model in Figure 14 have the following rules:

1. If activity A is executed, with x > 3 and y == "Sam", activity B should also be executed either before or after A is executed.

2. An execution of activity B with x==3 must be followed by activity C
3. Activity C should not be executed if it has an x value greater than 3.

Figure 15: Multiple constraint result obtained from the Data Aware Declare Replayer

The 2 results show slightly different optimal alignments with different fitness values. Figure 15a shows the result where the cost function assigns a higher cost to deviations associated to control flow over deviations associated to data write operations. Figure 15b shows the result where the cost function assigns a higher cost to deviations associated to data write operations over control flow. In 15a, the alignment introduces a move in log for event B and C to avoid breaking the set rules. That is, a move in log for event B avoids activating the response constraint which must be fulfilled by an execution of event C. Event C should not be executed and hence the introduction of the move in log on event C. In 15b, instead of introducing log moves, the system introduced incorrect asynchronous moves for the same activities for the same reasons. Both results do not violate any of the 3 rules and hence the correctness of our solution.

4.2.2 Performance evaluation

We evaluate the performance of our solution in terms of execution times under different scenarios. We aim to see the feasibility of our solution under different stress levels. To do so, we ran a couple of experiments with different parameters and recorded the execution times. The experiments had the following objectives:

- How does the solution perform given different log sizes and probabilities of violating all constraints within an event log?

- How does the solution perform given different model sizes, i.e. with different number of constraints?

- How does the solution perform given different cost functions?

2 sets of experiments that cover the above objectives were conducted. In the first set, we ran several experiments using one Declare model with 5 constraints. For each experiment, event logs with the same probability of violations but different sizes were generated. The probabilities chosen were 0.25, 0.5, 0.75 and 1. The event log sizes chosen were 250, 500 and 750. All traces were generated with the same length of 20 events. Each experiment was run 2(one with a higher control flow cost and one with a higher data write operations cost) x 5 times. The experiments were run 5 times and their averages execution times were recorded in seconds.

Probability of violation				
	25%	50%	75%	100%
Traces				
250	293.44	667.43	749.58	1089.61
500	678.5	1224.75	1555.98	2338.1
750	948.26	1621.27	2340.96	3037.5

Table 7: Experiment results (in seconds) for a cost function with a higher control-flow cost value

Probability of violation				
	25%	50%	75%	100%
Traces				
250	484.086	705.92	1026.65	1122.22
500	1126.21	1521.03	1605.42	2532.62
750	1336.21	1724.53	2934.17	2916.69

Table 8: Experiment results (in seconds) for a cost function with a higher data variable cost value

From the above results, it is clear that event log size, probability of violation and cost function do influence the performance of our solution. Table 6 shows the results of the experiments carried out with a cost function that assigns more cost value to control-flow deviations. The numbers

31

show significant increases in execution times as the probability of violation increases and also as the log size increases. The same is true with the results shown in Table 7 where experiments were carried out using a cost function that assigns a higher value to data variable violations. Comparing the tables, looking at corresponding cells, it shows that all values in Table 7 are much higher than the values in Table 6. This shows that assigning a higher cost to deviations associated with data variables invites higher execution times.

A second set of experiments which was intended to show that an increase in in the number of constraints also increases the execution times was carried out. With the knowledge of the above conclusions, we generated an event log for a Declare model with 10 constraints, 25% probability of violation and 250 log traces. The result was slightly above 3600 seconds and it clearly showed that adding constraints also influence the performance of our solution.

5 Related Work

An alignment can show how an event log can be replayed on a process model and how to change the log to perfectly fit the model. The principle of alignment in conformance checking of multi-perspective process models has been successfully implemented in procedural models. [1] implemented an approach that aligns a BPMN model and an event log to show deviations and the degree of conformance. To find an alignment between an event log and a model, *moves in the log are related to moves in the model*. Each move represents an *execution step* which consists of an executed activity/event and an assignment to the related data attributes. A cost function is introduced for legal moves which depends on the specific model and process domain. A trace and a model can yield multiple alignments. The goal of the approach is to find an *optimal alignment,* that is an alignment with the lowest cost. The authors in [1] employed the use of the A* algorithm to find the optimal alignment. The A* algorithm finds the path with the least cost between two nodes in a direct graph with costs associated to nodes. To apply the A* algorithm, an opportune search space needs to be defined. Every node of the search space is associated to a different alignment which is a prefix of some complete alignment between an event log trace and the model. The source node is an empty set and the target nodes set includes every complete alignment.

The search for an optimal alignment while considering multiple perspectives require more computational time and memory. In order to minimize computational time, [4] proposed a divide and conquer approach of the same technique in [1]. The process model is split into smaller model fragments and *for each fragment a sublog projecting the initial event log onto the activities used in the fragment* is created and hence aligned separately. For a valid decomposition of a Petri net with data, Single Entry Single Exit (SESE) based strategy is used in this paper. In [11] the same A* algorithm is used to find the optimal alignment. The search space is built using only control flow perspective of all possible moves. A* finds the shortest by queueing and visiting nodes with the smallest cost from the start node as well as the one that has the shortest distance to the target (an underestimation function). A poor underestimation function will cause so many nodes to be visited thereby using too much time and memory. [10] introduced an underestimation function based on the *marking equation of the Petri net*. This improves the efficiency of the A* algorithm by avoiding nodes that makes the final state no longer reachable as detected by the marking equation. In [11], the optimal alignment is found by formulating and solving an Integer Linear

Programming (ILP) problem. That is, using the control flow result of the A* algorithm, the solution of the ILP problem will then assign values to the variables of the control flow alignment.

[7] discovered a problem with the approach in [11] of returning non-optimal alignments thereby giving misleading explanations to deviations. Instead of checking the different perspectives in sequence, i.e. control-flow perspective first then using data variables on the result, the authors introduced an approach aimed at balancing all the perspectives at once. [7] formulates the problem of finding an optimal alignment as a search problem in a directed graph and employ the A* algorithm to find the least expensive path in the graph. Instead of building the directed graph beforehand, the search space is built incrementally. Starting with an empty node, a set of control-flow successors is built by considering only the control-flow perspective. The control-flow successors are not part of the search space. They are then augmented with the variable's write operations (data perspective). The augmentation process is defined as a multi integer linear problem (MILP) because the values of the variables need to be chosen that do not violate any condition and the aim is to minimize the cost of the deviations. If no solution is found for the MILP, no alignment is created, and the successor is discarded. If a solution is found for the MILP, i.e if an augmentation exists, a valid alignment is created, the cost is computed, and the alignment is added to the priority queue. When all successors have been identified, an alignment associated with the lowest cost is picked from the head of the queue. If the alignment is a complete alignment, then it is returned as the optimal alignment. *Otherwise the node is expanded, and successors are added to the queue*. A complete alignment is an alignment such that ignoring all 'no move' symbols should give back the original log and model before alignment. This approach however has a drawback of higher computational costs, that is it needs more computational power, but the efficiency brought by balancing the multiple perspectives when checking for conformance is of utmost importance.

As mentioned earlier, the above approaches work well for procedural models, e.g. Petri nets and BPMN. With declarative models, the constraints need to be encoded into another format that is easier to check for violations. Finite state machines are used in [20]. The conformance checking approach is based on the formalization of business constraints as *first-order linear temporal logic* (FOLTL) rules which are translated into finite state machines for *dynamically reasoning on*

partial, evolving traces. In order to monitor FOLTL, FO automata are built with *transactions labeled with first order formulas while the states contain data structures to smartly keep track of data.* The approach is used for evolving traces to quickly detect deviations at runtime. Instead of just stating whether a trace is good or bad, the authors needed to determine whether an evolving state has already been violated permanently or temporarily or is temporarily or permanently successful. That is for each transition, if it is the last activity and the current state is a final state or if it's not the last activity but, the current state is final and there is no transition reaching a *non-final* state from the current state then it is labeled as permanently successful. If the transition is the last activity and the current state is non-final or if is not the last activity, the current state is non-final and there exists no transition to a final state from the current state, then the trace is labeled as permanently violated. If the trace has not reached the last activity, the current state is final and there exist a transition from the current state to a non-final state then the trace is labeled as temporarily successful. If the trace has not reached the last activity, the current state is non-final and there exist a transition to a final state then the trace is labeled as temporarily violated. This approach was not used in the context of aligning data-aware declare models. At attempt at using finite state machines in the alignment of declarative models is found in [8]. Just like in the above-mentioned approaches for aligning procedural languages, [8] proposed an alignment-based conformance checking approach for declarative models. The approach also tries to replay activities in an event log against a Declare model by labeling moves in log where the move is only recognized by the log or move in model where a move is only recognized by the model and move in both. To determine the transitions, the Declare model is represented as final state automata for each constraint. Also, legal moves are associated with costs and the A* algorithm is employed to find the optimal alignment. However, this approach only recognizes the control flow perspective ignoring data, time and resources.

[21] introduces algorithms for conformance checking between an event log and an MP-Declare model. Generally, the proposed approach iterates through all the traces in the input log and for each constraint of the MP-Declare model, computes violations and fulfilments. Using a different algorithm for every constraint type in the MP-Declare model, the idea is to iterate through each event in the trace and for each, call operations that determines if an activation of a constraint occurs or a fulfilment of a pending activation occurs, or a violation of a constraint occurs. A closing

operation is also called at the end of the trace that converts all the pending activations into violations.

A constraint-based approach to conformance checking is proposed by [2]. It includes the modeling of declarative models as constraint satisfaction problems (CSPs) and a set of global constraints implemented through filtering rules is proposed. CSPs are transformed into Max-CSPs for the diagnosis process. The approach does not only provide conformance checking based on control-flow but also considers data. To solve the conformance checking problem through constraint programming, it needs to be modelled as a CSP. A CSP $P = (V, D, Ccsp)$ where V is a set of variables, D is the domain of variables and Ccsp is a set of constraints, control and data constraints, between variables such that each constraint represents a relation between a set of variables and specifies the allowed combinations of values for these variables. To solve a CSP, one needs to assign values to its variables. If the assignment of values to variables satisfies all its constraints, then the solution is considered to be feasible. A CSP is said to be feasible if there exists at least one related feasible solution which however is equivalent to compliancy. Therefore, the problem of checking conformance of an event log trace against a process model is equivalent to checking the feasibility of the CSP related to the model when instantiating its variables. If the instantiation is feasible then the trace is compliant otherwise it is non-compliant. This approach is different from our proposed approach because it focusses only on determining compliance without providing more diagnostic information. The approach does not also use alignments.

6 Conclusion and Future Work

In this thesis we presented an alignment-based data-aware conformance checking approach for declarative models using event logs. We used an extension of the Declare language to represent Declare models with data and our approach is an extension of the approach in [8] by including the data perspective. We showed how event log traces can be aligned with Declare models with data using a combination of Finite state automata, Integer Linear Programming and the A* algorithm. Our approach shows the degree of conformance in terms of fitness as well as exactly where in each trace, deviations occur. With the ability to specify a cost function, analysts can specify exactly which type of violation is more important than others. However, the choice of cost function can have an impact on the overall performance of the approach. Number of constraints, event log sizes and the probability of violation in each event log also influences the performance of our approach.

Our approach was implemented as a ProM plugin and evaluated using synthetic event logs. Despite efforts to improve the efficiency of the approach, the process uses too much resources and hence computation intensive. However, it is still feasible to use this approach with less complex models and smaller event logs.

For future work, the following points can be addressed:

- The implementation of the approach can be streamlined to improve on performance.
- We did not consider other parts of Declare models with data. We only focused on the activation condition. It is however possible to extend the implementation to include other parts such as the correlation condition and time constraints.
- Other ways of improving the efficiency of the A* algorithm, such as decomposition, can be used to improve the overall performance of the approach.
- We only showed the degree of conformance in terms of fitness, however other dimensions such as precision and generalization can also be calculated from our solution.

7 References

[1] M. de Leoni, W. M. van der Aalst and B. F. van Dongen, "Data- and Resource-Aware Conformance Checking of Business Processes," *Business Information Systems,* pp. 48-51, 2012.

[2] D. Borrego and I. Barba, "Conformance checking and diagnosis for declarative business process models in data-aware scenarios," *Expert Systems with Applications,* pp. 5340-5352, 2014.

[3] W. M. P. van der Aalst, W. Pesic and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D,* pp. 99-113, 2009.

[4] M. de Leoni, J. Munoz-Gama, J. Carmona and W. M. van der Aalst, "Decomposing Alignment-Based Conformance Checking of Data-Aware Process Models," in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, 2014.

[5] A. Adriansyah, J. Munoz-Gama, J. Carmona, W. van der Aalst and B. van Dongen, "Alignment based precision checking," *Process Management Workshops 2012, Lecture Notes in Business Proceedings of Business Information Processing,* vol. 132, no. Springer Verlag, Berlin, pp. 137-149, 2013.

[6] M. Dumas, M. La Rosa, J. Mendling and H. A. Reijers, "Process Intelligence," *Fundamentals of Business Process Management,* pp. 353-383, 2013.

[7] F. Mannhardt, M. de Leoni, H. A. Reijers and W. M. P. van der Aalst, "Balanced multi-perspective checking of process conformance," *Computing,* vol. 98, no. 4, pp. 407-437, 2015.

[8] M. de Leoni, F. M. Maggi and W. M. van der Aalst, "An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data," *Information Systems,* pp. 258-277, 2015.

[9] W. van der Aalst, A. Adriansyah and B. van Dongen, "Replaying history on process models for conformance checking and performance analysis," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery,* pp. 182-192, 2012.

[10] A. Adriansyah, B. van Dongen and W. van der Aalst, "Memory-efficient alignment of observed and modeled behavior," *Technology report, BPMcenter.org. BPM Center Report BPM-13-03,* 2013.

[11] 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.," *The 11th international conference on business process management (BPM'13), LNCS,* vol. 8094, no. Springer, pp. 113-129, 2013.

[12] "ProM website," [Online]. Available: http://www.promtools.org.

[13] H. Verbeek, J. Buijs, B. van Dongen and W. van der Aalst, "Xes, XESame, and Prom 6," *Information Systems Evolution,* vol. 72, pp. 60-75, 2011.

[14] C. Di Ciccio, M. L. Bernardi, M. Cimitile and F. M. Maggi, "Generating Event Logs Through the Simulation of Declare Models," *Lecture Notes in Business Information Processing,* pp. 20-36, 2015.

[15] F. M. Maggi, A. J. Mooij and W. M. van der Aalst, "User-guided discovery of declarative process models," *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM),* 2011.

[16] "Integer Programming," [Online]. Available: http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf.

[17] U. Bhat, "Runtime Monitoring of Data-Aware business rules with Integer Linear Programming," in *Masters Thesis, University of Tartu*, 2016.

[18] "LP Solver," [Online]. Available: http://lpsolve.sourceforge.net/5.5/.

[19] V. Skydanienko, "Data-aware Synthetic Log Generation for Declarative Process Models," *Masters thesis, University of Tartu,* 2018.

[20] R. De Masellis, F. M. Maggi and M. Marco, "Monitoring data-aware business constraints with finite state automata," *Proceedings of the 2014 International Conference on Software and System Process - ICSSP 2014,* 2014.

[21] A. Burattin, F. M. Maggi and A. Sperduti, "Conformance checking based on multi-perspective declarative process models," *Expert Systems with Applications,* pp. 194-211, 2016.

[22] B. Kitchenham, "Procedures for performing systematic reviews," in *Keele vol. 33*, UK, Keele University, 2004, pp. 1-26.

[23] W. Song, H.-A. Jacobsen, C. Zhang and X. Ma, "Dependence-Based Data-Aware Process Conformance Checking," *IEEE Transactions on Services Computing,* pp. 1-1, 2018.

[24] "XES-standard," [Online]. Available: http://www.xes-standard.org.

[25] "Declare Checker," [Online]. Available: https://svn.win.tue.nl/repos/prom/Packages/DeclareChecker/.

Appendix
I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Clive Tinashe Mawoko**,

(*author's name*)

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

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Aligning data-aware declarative process models and event logs,
(*title of thesis*)

supervised by **Prof. Fabrizio Maria Maggi**.
(*supervisor's name*)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Clive Tinashe Mawoko
21/05/2019