ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering - DISI
Two years Master Degree in Computer Engineering

Mining declarative process models with quantitative temporal constraints

Supervisor:
Prof. Federico Chesani
Correlator:
Prof. Paola Mello

Candidate
Orazio Cotroneo

Session II

Academic year 2020-2021

Contents

Introduction

Process mining [1] is a family of techniques aimed at facilitating the analysis of business models. There are two types of process models, imperative and declarative. Imperative process models describe step by step what is expected to happen in a process. Declarative process models are suitable to describe loosely structured processes. The discovery and analysis of business process models is based on data in a form of event logs.

Overall, declarative models support the analysis on the control-flow of a business process, but not on other aspects such as quantitative time and data. These limitations are usually caused by the logic used to define the language, as well as the complexity of the task.

The most widely used language to describe declarative process models is Declare [2], based on a set of templates which are defined using Linear Temporal Logic (LTL). Considering time as a linear timeline, LTL is used to describe properties of a system that vary over it. Declare templates exploits the properties of LTL by defining constraints between actives of a business process in a qualitative time approach.

This work aims to enhance declarative process models and enable quantitative temporal reasoning; and then it leverages on discovery algorithms to find the enriched model.

The first part of this thesis consists of identifying the approach to enrich the Declare templates with the notion of quantitative time. This starts from the analysis of the structure and the semantics of the Declare templates, and the possible alternative paths to enrich them. Once the semantics of the enriched templates is defined, the focus shifts in defining how they are calculated during the discovery phase.

The algorithms used for the discovery of the declarative process model are presented by Palmieri in *Learning Declarative Process Models from Positive and Negative Traces* [3]. These algorithms have been inspired by the learning Disjunctive Normal Form (DNF) and the learning Conjunctive Normal Form algorithms presented by Mooney in *Encouraging experimental results on learning CNF* [4].

These algorithms are consequently adapted to support the discovery of the enriched models. These changes are discussed in chapter 4, along with the experimental results, describing the obtained enriched models.

The last chapter of this work features a series of possible additions related to some of the Declare templates that were deemed out of the scope of this work. Performance improvements of the existing algorithms are also discussed in the final chapter.

1 Background

This chapter contains an introduction to the technologies that are fundamental to the work explored in this thesis. First, it introduces the basic concepts of process mining and the differences between imperative and declarative process modelling. Second, it describes the Declare templates with an overview of Linear Temporal Logic. Finally, it gives an overview of the Prolog programming language.

1.1 Process mining

Process mining [1] can be defined as the automated art of deeply understanding a business process beginning with the analysis of transactional data in the form of logs. Process mining aims to shift from a confidence/observational-based approach to a reliable and unbiased evidence-based approach. It makes it possible to map, explore and understand all the possible paths within a business process. It highlights possible bottlenecks and addresses issues through non-compliance perspective.

Process mining creates a fundamental bridge between data mining and business process modelling, with event data being the focus point. In fact, process mining assumes every process can be recorded as a series of events, with each event referring to an occurrence of an activity. Additional information can be logged in the events, such as the point in time at which the activity was executed. This is referred to as timestamp. Therefore, it is important to treat event data as first-class citizens, rather than something that is non-essential or decoration to the system. Events should have a well-defined structure and they should be logged automatically and reliably. A complete set of well-defined data, considering its structure and clear semantic, can be used to meaningfully model and analyse a process.

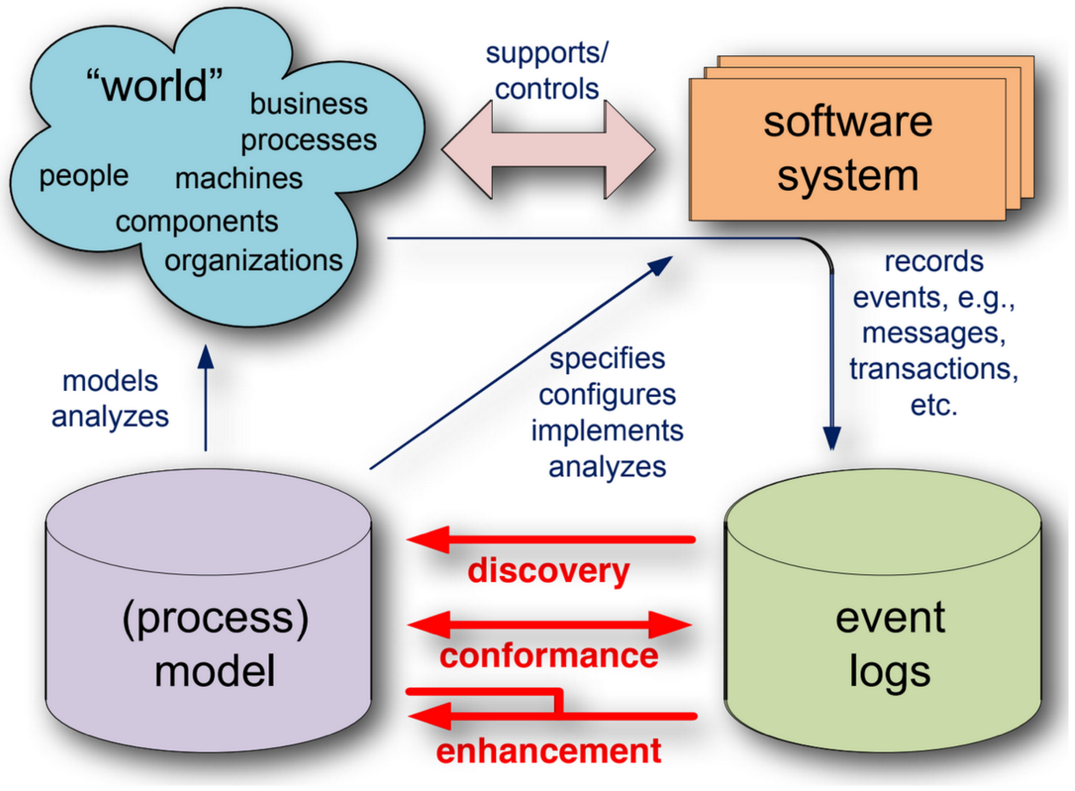

Figure 1.1: Positioning of the three main types of process mining [1]: (a) discovery, (b) conformance checking, and (c) enhancement.

According to [1], there are three basic process mining techniques: Discovery, Conformance Checking and Enhancement which deal with the connections between data events and models.

The Discovery technique creates a model based on event logs. The outcome of Discovery will be "actionable process knowledge" in the form of a model. The Conformance Checking technique checks if the logs are conforming with the model and vice versa. This validates the alignment between the model and the event data. The Enhancement technique aims to improve or modify the existing model, by comparing it to a set of new event data.

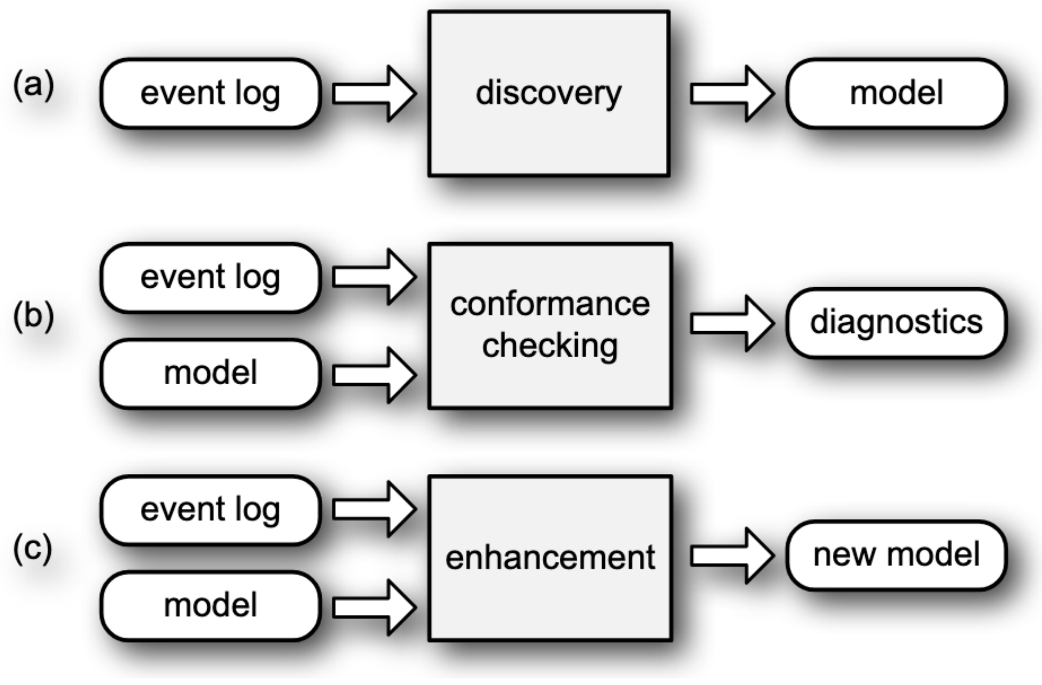

Figure 1.2 The three basic types of process mining explained in terms of input and output [1]: (a) discovery, (b) conformance checking, and (c) enhancement.

1.1.1 Process mining guiding principles

There are six guiding principles in process mining as described in the process mining manifesto [1].

GP1: Event logs should aim at the highest possible quality level

The quality of all process mining activities starts with the collection of event logs and the quality of their results depend heavily on the quality of the input events. There are different criteria to evaluate the quality of event data: trustworthy, complete, safe and semantically well-defined - for example, that the recorded events actually happened, and their attributes are correct (trustworthy); no events are missing (complete); they have a well-defined semantic and privacy and security were respected when they were recorded (safe). Based on these criteria, it is possible to score event log maturity levels from five stars indicating excellent quality to one star for poor quality.

GP2: Event logs extraction should be guided by questions

It is only possible to extract meaningful and relevant event data with concrete questions. Therefore, before applying any process mining technique, it is necessary to select the type of cases to be analysed. This should be driven by the questions that need to be answered.

GP3: Concurrency, choice and other basic control-flow constructs should be supported

As described in [1], it is important to support at least the basic workflow patterns: concurrency, sequence, parallel routing (AND-splits/joins), choice (XOR-splits/joins), and loops. Otherwise, resulting models may be badly underfitted or extremely complex. Support for OR-splits/joins should also be considered, because they provide a representation of inclusive decisions and partial synchronizations.

GP4: Events should be related to model elements

Process mining is not only limited to control-flow discovery, but it also includes conformance checking and enhancement. The latter relies on the relationship between elements in the model and events in the log to make different types of analysis.

GP5: Models should be treated as purposeful abstractions of reality

Models derived from event data represent a particular view of reality. Given an event log, there may be multiple views that are useful to a particular type of user. Moreover, these models may have different levels of granularity and abstraction depending on the needs of its users. Visualisation of process models should be designed in such a way that the information represented is easy to interpret by its intended audience.

GP6: Process mining should be a continuous process

Process mining should be seen as a continuous process based on historical and real-time data providing actionable information. Because processes change as they are being analysed, process mining should not be seen as a one-time activity. Instead, users should be encouraged to look at them regularly.

1.2 Imperative vs Declarative process modelling

There are two main approaches to process modelling: the imperative and the declarative. In imperative models, the succession of events is explicitly visible in the model and all the possible interactions between each activity are specified. This makes the models straightforward to understand. However, this also implies that for loosely structured processes the number of variants might make the model difficult to read. These differences are further explored in more detail using an example use case where a person is travelling by train.

The Business Process Model and Notation[1] [5] (BPMN) model in Figure 1.3 represent a single variant[2] model of a traveller arriving at a train station, purchasing the ticket at self-service ticket machine, scanning it at the ticket barrier, and then boarding a train.

[1] Workflow Net, which are the extension of Petri Net, and Business Process Model and Notation are used to describe imperative models in process mining.

[2] A variant is a possible path from the start to the end event.

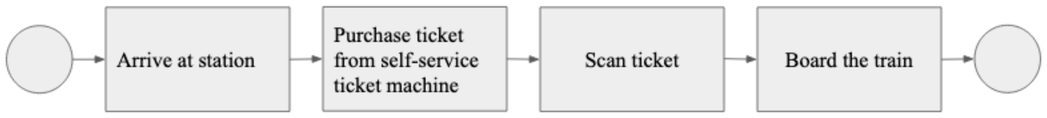

Figure 1.3: Example of a BPMN model.

The BPNM model, in Figure 1.4, has two possible variants at the purchase stage. The traveller can either purchase a ticket from a self-service ticket machine or from the ticket office.

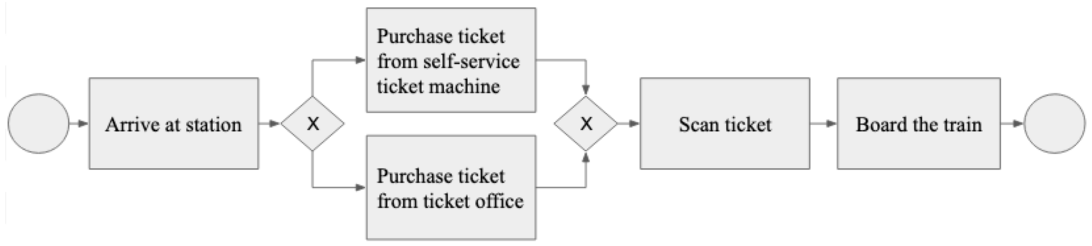
Arrive at station
X
Purchase ticket from self-service ticket machine
Purchase ticket from ticket office
X
Scan ticket
Board the train

Figure 1.4: Example of a BPMN model.

To include cases where the traveller has already purchased the ticket either via a website or a mobile application, either before or after arriving at the station, the model should be enriched with more variants.

In contrast, Declarative models [6] specifies a set of constraints that will have to be fulfilled during the process execution. The order of events is implicitly specified by the constraints - this makes it more difficult to understand the different variants but allows for more flexibility.

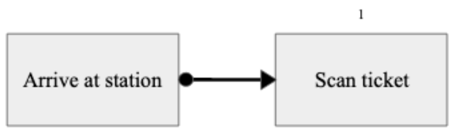

Figure 1.5 Example of a Declare model.

The Declare model in Figure 1.5 can determine if a traveller has arrived at a station and has eventually scanned the ticket but cannot answer the question “How much earlier does a traveller arrive at the station, before they scan their ticket?”. The quantitative information about how much time passes between the traveller arriving at the station and the ticket being scanned is currently missing. Although the information itself can be extracted from the timestamps in the logs, Declare does not support quantitative time modelling.

1.3 Process modelling using Declare

Declare is a constraints-based declarative process modelling language. It has two types of constraints, the unary constraints (also known as existence constraints) and the binary constraints (also known as relation constraints). The existence constraints deal with only one activity, while the relation constraints deal with two activities: the activation and the target activity. The activation represents an event that will constrain the subsequent events; it is eventually associated with fulfilment or failure, based on the constraint and the occurrence of the target event.

In the example in Figure 1.5, there are two constraints represented: a relation constraint between “Arrive at station” as the activation activity, and “Scan ticket” as the target activity, and an existence constraint on “Scan ticket”. The existence constraint is asserting that the ticket can only be scanned exactly once.

1.3.1 Declare templates

In her work [2], Pesic describes four main groups of templates: existence, relation, negation and choice templates; and formalises them using Linear Temporal Logic (LTL). In this work, the templates concerning the absence of activities are listed in the non-existence group. Table 1 lists the Declare templates divided per group.

Template group	**Templates**
Existence templates	*init(A)*, *last(A)*, *existence(A)*, *existence2(A)*, *existence3(A)*, *exactly1(A)* and *exatcly2(A)*.
Non-existence templates	*absence(A)*, *absence2(A)* and *absence3(A)*.
Choice templates	*choice(A,B)* and *exclusive_choice(A,B)*.
Relation templates	*responded_existence(A,B)*, *co-existence(A,B), response(A,B)*, *precedence(A,B)*, *succession(A,B)*, *alternate_response(A,B)*, *alternate_precedence(A,B)*, *alternate_succession(A,B)*, *chain_response(A,B)*, *chain_precedence(A,B) and chain_succession(A,B).*
Negation templates	*not_responded_existence(A,B)*, *not_co-existence(A,B)*, *not_response(A,B)*, *not_precedence(A,B)*, *not_succession(A,B)*, *not_chain_response(A,B)*, *not_chain_precedence(A,B)* and *not_chain_succession(A,B)*.

Table 1: Groups of Declare templates.

Linear Temporal Logic (LTL)

Given its declarative nature, Linear Temporal Logic [7] is used for the formal specification of constraint templates in declarative process models, as presented by Pesic in [2]. LTL is a logic, or formalism, used for specifying properties of a system that vary with time. Whilst propositional logic deals with statements that are true or false at a given state, LTL extends this concept over time as the system moves through a sequence of states. LTL considers time as a linear timeline in which a series of events occur one after the other at a specific point in time and there are no alternative timelines or branching time.

Formulae of LTL are built from a set of finite propositional variables[3], logical operators and temporal modal operators. A well-formed LTL formula, as described by Pesic in [2], is a function p over a subset E of all possible events, and it is defined as p:$E^*\rightarrow\{\text{true, false}\}$. Let $\sigma \in E^*$ be a trace, if p is a well-formed formula and it holds that p(σ) = true then p satisfies σ, denoted by $\sigma \vDash$ p. In addition, if p and q are well-defined formulas, then true, false, !p, p $\wedge$ q, p $\vee$ q, $\Diamond$p, $\Box$p, $\bigcirc$p, pUq and pWq are well-formed formulas as well.

[3] A propositional variable is a variable that can either assume a true or false value.

The semantics of LTL are defined as:

proposition: $\sigma \vDash$ e if and only if e = $\sigma[1]$, for e $\in$ E.

not (!): $\sigma \vDash$!p if and only if not $\sigma \vDash$ p.

and (∧): $\sigma \vDash$ p ∧ q if and only if $\sigma \vDash$ p and $\sigma \vDash$ q.

or (∨): $\sigma \vDash$ p ∨ q if and only if $\sigma \vDash$ p or $\sigma \vDash$ q.

next (◯): $\sigma \vDash$ ◯p if and only if $\sigma^{2\rightarrow} \vDash$ p.

until (*U*): $\sigma \vDash$ pUq if and only if ($\exists_{1\leq i\leq n}$:($\sigma^{i\rightarrow} \vDash$ q ∧ ($\forall_{1\leq j\leq i}$ $\sigma^{j\rightarrow} \vDash$ p)).

It is also possible to use the following abbreviations:

implication (p ⟹ q): for !p ∨ q.

equivalence (p ⟺ q): for (p ∧ q) ∨ (!p ∧ !q).

true (true): for p ∨ !p.

false (false): !true

eventually(◊): for ◊p $=$ true$U$p

always(□): for □p $=$!◊ !p.

weak until (*W*): for pWq $=$ (pUq) ∨ (□p).

Existence templates

The existence templates express the cardinality, or the position of a given activity in a trace. The non-existence templates express the absence of a given activity in the trace. Table 2 lists LTL expression of existence and non-existence templates.

The *existence(A)*, *existence2(A)* and *existence3(A)* templates specify the fewest occurrences for activity A. These templates allow at least one, at least two and at least three activities in the trace respectively.

The *absence(A)*, *absence2(A)* and *absence3(A)* templates specify the greatest number of occurrences for activity A. These templates allow no activity, at most one and at most two activities in the trace respectively.

The *exactly1(A)* and *exactly2(A)* templates specify the exact number of occurrences for the activity A. These templates allow exactly one and exactly two activities in the trace respectively.

The *init(A)* and *last(A)* specify the position of activity A within a trace. The position has to be the first or last in the trace respectively.

Template	**LTL expression**
existence(A)	$\Diamond A$
existence2(A)	$\Diamond(A \land \bigcirc(\mathit{existence(A)}))$
existence3(A)	$\Diamond(A \land \bigcirc(\mathit{existence2(A)}))$
absence(A)	!*existence(A)*
absence2(A)	!*existence2(A)*
absence3(A)	!*existence3(A)*
exactly1(A)	*existence(A)* ∧ *absence2(A)*
exactly2(A)	*existence2(A)* ∧ *absence3(A)*
init(A)	A
last(A)	$\Diamond(A \land \bigcirc!\mathrm{true})$

Table 2: LTL expression of existence and non-existence templates [2].

Choice templates

The choice templates express the presence of either both or one of the two activities in a trace. Table 3 lists LTL expression of choice templates.

The *choice(A,B)* template states that either activity A or B has to occur, but they can both occur. The *exclusive_choice(A,B)* strengthens this condition by asserting that only one of the two activities has to happen within a trace.

Template	**LTL expression**
choice(A,B)	$\Diamond A \vee \Diamond B$
exclusive_choice(A,B)	$(\Diamond A \vee \Diamond B) \wedge !(\Diamond A \wedge \Diamond B)$

Table 3: LTL expression of choice templates [8].

Relation templates

The relation templates express the desired relative position of an activity regarding another activity within the trace. Table 4 lists LTL expression of relation templates.

The *responded_existence(A,B)* states that if an activity A occurs, then an activity B has to occur, either before or after the occurrence of activity A. The *co-existence(A,B)* has two activation points, A and B. If activity A occurs then activity B has to occur, and if activity B occurs then activity A has to occur.

The *response(A,B)* states that every time activity A occurs, then activity B has to occur after it. The *precedence(A,B)* states that every time activity B occurs then activity A has to occur before it. The *succession(A,B)* states that both *response(A,B)* and *precedence(A,B)* have to hold in order for it to hold.

The *alternate_response(A,B)* strengthens the condition relative to the position of the activation and target activities compared to *response(A,B)*, stating that between the occurrences of the activities A and B, there cannot be another activity A. Likewise *alternate_precedence(A,B)* strengthens the condition relative to the position of the two activities stated in *precedence(A,B)*. The *alternate_succession(A,B)* states that both *alternate_response(A,B)* and *alternate_precedence(A,B)* have to hold for it to hold.

Template	**LTL expression**
responded_existence(A,B)	$\lozenge A \Rightarrow \lozenge B$
co-existence(A,B)	$\lozenge A \Leftrightarrow \lozenge B$
response(A,B)	$\square(A \Rightarrow \lozenge B)$
precedence(A,B)	$!B\ W\ A$
succession(A,B)	*response(A,B)* ∧ *precedence(A,B)*
alternate_response(A,B)	*response(A,B)* ∧ □(A ⟹ ◯(*precedence(A,B)*))
alternate_precedence(A,B)	*precedence(A,B)* ∧ □(B ⟹ ◯(*precedence(A,B)*))
alternate_succession(A,B)	*alternate_response(A,B)* ∧ *alternate_precedence(A,B)*
chain_response(A,B)	*response(A,B)* ∧ □(A ⟹ ◯B)
chain_precedence(A,B)	*precedence(A,B)* ∧ □(◯B ⟹ A)
chain_succession(A,B)	*chain_response(A,B)* ∧ *chain_precedence(A,B)*

Table 4: LTL expression of relation templates [2].

The *chain_response(A,B)*, *chain_precedence(A,B)* and *chain_succession(A,B)* further strengthens the condition relative to the position of the activation and target activities, stating that there are not occurrences of any other activity between A and B.

Negation templates

The negation templates are the negated version of the relation templates. They verify that given the presence of the activation activity, the target activity is absent. For example, the *not_responded_existence(A,B)* states that if activity A occurs then activity B does not have to occur, either before or after the occurrence of activity A.

Table 5 lists LTL expression of negation templates.

Template	**LTL expression**
not_responded_existence(A,B)	$\lozenge A \Rightarrow !(\lozenge B)$
not_co-existence(A,B)	*not_responded_existence(A,B)* ∧ *not_responded_existence(B,A)*
not_response(A,B)	$\square(A \Rightarrow !(\lozenge B))$
not_precedence(A,B)	$\square(\lozenge B \Rightarrow !A)$
not_succession(A,B)	*not_response(A,B)* ∧ *not_precedence(A,B)*
not_chain_response(A,B)	$\square(A \Rightarrow \bigcirc(!B))$
not_chain_precedence(A,B)	$\square(\bigcirc B \Rightarrow !A)$
not_chain_succession(A,B)	*not_chain_response(A,B)* ∧ *not_chain_precedence(A,B)*

Table 5: LTL expression of negation templates [2].

1.3.2 Prolog

The programming language used in this thesis is Prolog. Exploiting its declarative nature to determine patterns, Prolog was deemed to be suitable to identify constraints within the traces[3]. Prolog, or PROgramming in LOGic, is a logic programming language designed by Colmerauer and Roussel in 1972[4], and it is based on the exploratory work, by Kowalski in [9], to use the properties of Horn clauses in the context of logic programming.

A program in Prolog is defined as a set of Horn clauses represented as facts, rules and goals. Taking the 2020 UEFA European Football Championship as an example, a Prolog program might look as follow:

```
team(italy).  /* Example of fact */

winner(italy, euro2020). /* Example of fact */

champion(X,Y) :- team(X), winner(X,Y). /* Example of rules */

:- champion(X, euro2020). /* Example of goal */
```

The execution of a Prolog program starts from a query. For further information on Prolog and the resolution strategy adopted by the Prolog engine, refer to [10] and [11].

[4] In [12], Colmerauer and Roussel describe the evolution of Prolog.

Swi-Prolog[13] is used as developing tool as it offers a rich set of predefined features and libraries. The Constraint Logic Programming over Finite Domain library CLP(FD) [14], is used to reason over integers using declarative integer arithmetic.

2 Introduction to the DNF and CNF learning algorithms

The algorithms used to discover quantitative time constraints are based on the work on learning declarative process models from positive and negative traces as discussed by Palmieri [3]. Her work expands on conjunctive normal form (CNF) and disjunctive normal form (DNF) algorithms from Mooney [4].

In this chapter, to give a general understanding of the evolution of the two algorithms, both Palmieri's and Mooney's algorithms are discussed.

In [4], Mooney compared three inductive learning systems using different representations for concepts: conjunctive normal form, disjunctive normal form and decision tree. In the five natural datasets tested, the CNF learner consistently obtained greater or equal classification accuracy, ran faster and produced fewer complex concepts.

For the purpose of this work, only CNF formulae and DNF formulae are presented. The two algorithms learn first-order Horn clauses; however, the basic algorithms are heuristic covering algorithms for learning DNF or CNF.

2.1 Disjunctive Normal Form

The algorithm comprises of two nested cycles. The outer loop takes in as input the list of positive and negative examples, removes the positive examples that it covers and ends when the list of positive examples is empty. Before calling the inner loop, a copy of the sets is made. The inner loop instead removes the negative examples that it covers and ends when the list is empty. Results are then returned as a disjunctive normal form.

DNF Learning Algorithm: PFoil [4]

```
Let Pos be all the positive examples.
Let DNF be empty.
Until Pos is empty do:
    Let Neg be all the negative examples.
    Set Term to empty and Pos2 to Pos.
    Until Neg is empty do:
        Choose the feature-value pair, L, that maximizes DNF-gain(L,
        Pos2,Neg2)
        Add L to Term.
        Remove from Neg all examples that do not satisfy L.
        Remove from Pos2 all examples that do not satisfy L.
    Add Term as one term of DNF.
    Remove from Pos all examples that satisfy Term.
Return DNF

Function DNF-gain(L,Pos,Neg)
    Let P be the number of examples in Pos and N the number of examples in Neg
    Let p be the number of examples in Pos that satisfy L.
    Let n be the number of examples in Neg that satisfy L.
    Return p*(log(p/(p+n)) - log(P/(P+N)))
```

CNF Learning Algorithm: PFoil-CNF [4]

```
Let Neg be all the negative examples.
Let CNF be empty.
Until Neg is empty de:
    Let Pos be all the positive examples.
    Set Clause to empty and Neg2 to Neg,
    Until Pos is empty de:
        Choose the feature-value pair, L, that maximizes CNF-gain(L,Pos,Neg2)
        Add L to Clause.
        Remove from Pos all examples that satisfy L.
        Remove from Neg2 all examples that satisfy L.
    Add Clause as one clause of CNF.
    Remove from Neg all examples that do not satisfy Clause.
Return(CNF)

Function CNF-gain(L,Pos,Neg)
    Let P be the number of examples in Pos and N the number of examples in Neg.
    Let p be the number of examples in Pos that do not satisfy L.
    Let n be the number of examples in Neg that do not satisfy L.
    Return n*(log(n/(p+n)) - Iog(N/(P+N)))
```

2.2 Conjunctive Normal Form

The main difference with the previous algorithm is that CNF returns results as conjunctive normal form. Whilst DNF learns terms until all positive examples are covered, CNF learns clauses until all the negatives are removed. Whilst DNF learns clauses one literal at a time until all negatives are removed, CNF learns clauses one literal at a time until all positives are covered.

2.3 Learning Declarative Process Models from Positive and Negative Traces

Starting from Mooney's work [4], Palmieri [3] has adapted his DNF and CNF algorithms to process mining.

2.3.1 Disjunctive Normal Form

To avoid redundant terms, terms returned by the inner cycle are only added to it if they cover at least one positive trace. The list returned by *remove_satisfying_traces* is compared to the old one. If they are the same, then no positive traces have been removed and therefore no trace satisfies the new term. The term is then discarded.

Implementation of the Learning Declarative Process Models from Positive and Negative Traces Algorithm [3]

```
start :-
      get positive traces(Pos),
      get negative traces(Neg),
      outer cycle(Pos, Neg, Model, PosNotCovered, NegNotExcluded).
outer_cycle([ ], Neg, Model, PosNotCovered, NegNotExcluded).
outer_cycle(Pos, Neg, Model, PosNotCovered, NegNotExcluded) :-
      inner_cycle(Pos, Neg, Term, NewNegNotExcluded),
       remove_satisfying_traces(Pos, Term, PosLeft),
      (PosLeft == Pos → PosNotCovered is Pos; Pos is [ ]),
      outer_cycle(PosLeft, Neg, [Term|Model], PosNotCovered,
         NewNegNotExcluded).
inner_cycle(Pos, [ ], Term, NewNegNotCovered).
inner_cycle(Pos, Neg, Term, NewNegNotCovered) :-
      choose_constraint(Pos, Neg, Term, NewConstraint),
      remove_not_satisfying_traces(Pos, NewConstraint, PosLeft),
      remove_not_satisfying_traces (Neg, NewConstraint, NegLeft),
      inner_cycle(PosLeft, NegLeft, [NewConstraint|Term],
         NewNegNotCovered).
                                          %Enters here if choose constraint fails
inner_cycle(Pos, Neg, Term, NewNegNotCovered) :-
   NewNegNotCovered is Neg,
   Inner_cycle(Pos, [ ], Term, NewNegNotCovered).
```

Moreover, her algorithm allows for some traces not to be covered by the model to avoid having both positive and negative traces that are in contrast with the other ones. In Mooney's algorithm, the same trace can be included in both sets of positive and negative examples. As a result, it would be impossible to create a model that includes the positive trace while at the same time excludes the negative one. To overcome this, if the *choose_constraint* fails because there are no more constraints to be added; the negative traces left are not possible to exclude, so these are removed and saved in a variable: *NewNegNotCovered*. Likewise, if there are no more terms that can cover the remaining positive trace, they are discarded and saved to a variable. Once the model is returned, the discarded traces are printed to screen to inform the user that the process model was discovered only with part of the logs.

The *choose_constraint* function has three parameters: the current lists of positive examples, the current list of negative examples and the list of constraints already added to the term. It returns the constraint that will be added to the term by the inner cycle. The function searches for the best constraints based on the positive and negative traces. Starting from the first level of the hierarchy[5], it combines the templates with all the possible activities found in the log.

[5] Certain templates strengthen the condition of other templates, forming a hierarchy of templates. For example: *chain_response(A,B)* template strengthens *response(A,B)* by specifying that B must immediately follow A. The hierarchy of templates can be found in [3]. In the paragraph 4.2, the enriched hierarchy of template is described in detail.

The DNF choose_constraint function [3]

```
choose constraint(ListOfPositiveExamples, ListOfNegativeExamples, Term,
  NewConstraint) :-
    combine(FirstLevelOfHierarchy, Activities,
      GroundedFirstLevelOfHierarchy),
    specialize_existing_constraints(GroundedFirstLevelOfHierarchy, Term,
      ListOfPossibleCandidates),
    get_best(ListOfPossibleCandidates, ListOfPositiveExamples,
      ListOfNegativeExamples, NewConstraint).
```

The algorithm starts from the same set of constraints, and not from the ones that are already in the term because some branches of the hierarchy might have not been explored yet, so it is faster to remove the ones already present than to add the ones that are not.

Next, it checks which constraints are already in the term and descends the hierarchy to find the more specialized ones. It ends when the new set of constraints to choose from is complete. This set contains the first-level constraints that are not already present in the term and all the possible specializations of the ones that were chosen in previous iterations.

If the first constraint in the list is already present in the term, then the *findall* function called on the hierarchy of templates returns all its possible specialisations. The constraint candidates and the ones already added to the set are divided into two variables.

Once it has the complete list of constraints to choose from, it removes the duplicates and assigns each constraint a score, calculated through the DNF gain function. It then selects the constraint that has the highest score.

The DNF gain function from Mooney [4] calculates the gain of each constraint based on the numbers of positive P and negative N traces, and the number of positive p and negative n traces that satisfy the constraint. If the number of satisfied positive traces is 0, the function $\log_{10}(p/p + n)$ results in $-\infty$. This in Prolog generates an error. In [3], Palmieri overcome this by assigning a very low number to the gain; that assures the constraint will not be chosen. Instead, if the number of satisfied positive traces is bigger than 0, the higher the value of p is, the higher the gain. Considering two constraints that satisfy the same number of positive traces, then the gain is determined by the number of negative traces that satisfy each constraint. The higher the number of negative traces satisfied by the constraint, the lower the gain.

The dnf_gain function [3]

```
dnf_gain(Constraint, ListOfPositiveTraces, ListOfNegativeTraces) :-
    Let P be the number of examples in ListOfPositiveTraces,
    Let N be the number of examples in ListOfNegativeTraces,
    Let p be the number of examples in ListOfPositiveTraces that satisfy
      Constraint,
    Let n be the number of examples in ListOfNegativeTraces that satisfy
      Constraint,
    Return $p \times (\log_{10}(\frac{p}{p+n}) - \log_{10}(\frac{P}{P+N}))$.
```

2.3.2 Conjunctive Normal Form

The CNF algorithm from Palmieri [3] returns a model in the conjunctive normal form. As per the DNF, the CNF algorithm is an adaptation of Mooney's work [4]. Here, the outer cycle ends when all negative traces are excluded by the model, while the inner cycle ends when all positive traces satisfy the clause.

The inner cycle chooses the most convenient constraint, and it removes all traces that satisfy it from its local list of positive and negative examples. This is different from the DNF algorithm which removed all traces that did not satisfy the constraint. The main reason for this difference is that every clause needs to cover all positive traces, and as their constraints are in OR, the examples covered by one constraint are also covered by the whole clause too. In later iterations the focus is on the ones that are not yet covered, so the clause will not have redundant constraints. As a result, the model is wider.

The outer cycle removes the negative traces that do not satisfy the clause returned by the inner cycle. If any trace is removed, then the clause is added to the model in AND. As a result, the model is as restrictive as the most restrictive clause. If a clause does not cover a positive trace, then the model will not cover it.

As for the DNF algorithm, traces are excluded from the model instead of failing, if it is not possible to produce a model that describes them. The inner cycle removes the positive ones, while the outer cycle removes the negative ones, if the inner cycle fails to produce a clause that covers them.

The template hierarchy for the CNF algorithm is made of all the most specialized templates, since the constraints in the clauses are in OR. The hierarchy of templates is then scanned in reverse, starting from the templates that are the leaves of the tree and reaching the root nodes in later iterations.

The choose constraint function chooses the best constraint to add to the clause. It always starts from the most specialized constraints and then adds more general ones in later iterations.

The CNF choose_constraint function [3]

```
choose_constraint(ListOfPositiveExamples, ListOfNegativeExamples, Clause,
  NewConstraint) :-
    combine(LastLevelOfHierarchy, Activities,
      GroundedLastLevelOfHierarchy),
    generalize_existing_constraints(GroundedLastLevelOfHierarchy, Term,
      ListOfPossibleCandidatesToCombine),
    combine(ListOfPossibleCandidatesToCombine, Activities,
      ListOfPossibleCandidates),
    get_best(ListOfPossibleCandidates, ListOfPositiveExamples,
      ListOfNegativeExamples, NewConstraint).
```

The CNF constraint function has an extra combine function. This is necessary because in some subtrees of the hierarchy the parent node is a template with two variables and the child node only has one. This step substitutes the value with every possible activity, creating all the different constraints. Note that in the case where the template given to the combine function is already grounded, the function will simply return the template.

The CNF gain function from Mooney [4] calculates the gain of each constraint based on the numbers of positive P and negative N traces, and the number of positive p and negative n traces that satisfy the constraint. If the number of satisfied negative traces is 0, the function $\log_{10}(n/p + n)$ results in $-\infty$. This in Prolog generates an error. In [3] Palmieri overcame this by assigning a very low number to the gain; that assures the constraint will not be chosen. Instead, if the number of satisfied negative traces is bigger than 0, the higher that value is, the higher the gain. Considering two constraints that satisfy the same number of negative traces, then the gain is determined by the number of positive traces that satisfy each constraint. The higher the number of positive traces satisfied by the constraint, the lower the gain.

The cnf_gain function [3]

```
cnf_gain(Constraint, ListOfPositiveTraces, ListOfNegativeTraces) :-
    Let P be the number of examples in ListOfPositiveTraces,
    Let N be the number of examples in ListOfNegativeTraces,
    Let p be the number of examples in ListOfPositiveTraces that do not
      satisfy Constraint,
    Let n be the number of examples in ListOfNegativeTraces that do not
      satisfy Constraint,
    Return $n \times (\log_{10}(\frac{n}{p+n}) - \log_{10}(\frac{N}{P+N}))$.
```

As described by Palmieri in [3], the algorithm is more efficient if more negative traces are excluded by a clause. In addition, the model will become simpler because the number of constraints will be lower.

3 Time in process modelling

In this chapter, the concept of time in process modelling is explored starting from understanding the differences between qualitative and quantitative time to defining the approach to enrich Declare templates with quantitative time metrics.

In order to mine quantitative time, the main challenges to overcome are how to enrich the Declare templates with the concept of quantitative time metrics (Declare templates are formalised using propositional LTL over finite traces) and then how to adapt the DNF and CNF to accommodate the enriched templates definitions.

So that the right approach is defined to enrich declare templates with quantitative time metrics, it has been useful to identify and explore the systems that already deal with the same challenge. In recent years the necessity to consider data-aware constraints for loosely structured processes has arisen. In particular, the introduction of quantitative time for monitoring systems has been a field of research. Several monitoring systems have been analysed, such as BPath [15], MONPOLY [16][17], Giblin [18] and MuboconEC [19]. They all adopt quantitative time metrics but with different approaches. BPath [15] is based on XPath to query logs, the quantitative time is considered as the differences between timestamps associated to events. MONPOLY [16][17] and Giblin [18] both use extensions of the LTL. MONPOLY [16][17] is based on Metric first-order temporal logic to deal with quantitative time, while Giblin [18] is based on Timed Propositional Temporal Logic. MuboconEC [19] is a monitoring system defined using the Event Calculus. In MuboconEC [19] the quantitative time metrics are explicitly introduced using two variables, that are subject to arithmetic constraints.

3.1 Qualitative vs Quantitative time

Time has two characteristics associated to it, qualitative and quantitative[6]. The qualitative character of time refers to the position that a given event occupies in a series of events or trace.

Defining the system under analysis (the trace) is therefore essential to the definition of qualitative time itself. The position of the event is considered relative to the first or last event, or to any another event within the trace. To illustrate this, consider the sample events "buy the ticket" to book to visit a temporary exhibition at the Tate Modern in London, and the sample event "enter the exhibition" to show the ticket for entry the exhibition. It is clear that there is a qualitative time relation between the two sample events, since the "buy the ticket" event has to occur before the occurrence of the "enter the exhibition" event. Therefore, the qualitative aspect of time maps to "when" an event has to occur.

There is also another aspect of qualitative time which refers to the cardinality of an event. This corresponds to "how many times" a specific event has to occur. For example, considering the current Covid-19 pandemic, to ensure social distancing measures are in place, each ticket can only be used once to enter the exhibition. This implies that the sample event "enter the exhibition" has to happen only once.

The quantitative time character refers instead to the measurement of the duration of a time interval in which a specific event can or has to occur, corresponding to the

[6] Time has always been a subject of study in science, philosophy and religion. Time was referred by the ancient Greeks with two separate words: Chronos and Kairos. Chronos referring to the quantitative aspect of it, while Kairos referring to the qualitative part of it.

question “how long” do we need to wait to experience a certain event. Considering the validity of the ticket to be only four hours from the point of purchase, this implies that the “enter the exhibition” event has to happen within four hours of the “buy the ticket” event. The time interval in this case is four hours.

Declare templates are formalised using propositional LTL over finite traces. In Linear(-time) Temporal Logic, time is considered as a linear timeline, having one single realization of it and the trace is considered as series of events occurring one after the other at a specific point in time. Based on these premises the semantic of Declare templates has been formalised in terms of LTL in [2]. LTL is a logic that reasons with propositions qualified in terms of qualitative time. Therefore, enriching the Declare templates with quantitative time metrics has been a challenge due to the expressiveness power of the underlying logic.

3.2 Augmented Declare templates

The approach adopted to enrich Declare templates with quantitative time metric constraints is inspired by the work proposed by Montali for compliance monitoring in [19]. The templates have been annotated with two numerical values, Delay and Deadline. Delay is defined as the minimum time before an activity occurs, while Deadline is defined as the maximum amount of time in which an event can occur. Delay and Deadline together define the time interval in which an event can occur. In [20], Montali also introduces a formal representation of timed binary data-aware constraints.

To facilitate the reasoning in terms of duration of time associated to the occurrence of an event, Delay and Deadline values will predicate only on the existence of activities rather than on their absence. For this reason, the templates that are extended with quantitative time metrics are the existence and relation templates, and the choice template.

Existence, choice and relation templates come with a different set of problems when trying to calculate the quantitative timing metrics associated to them. Because the existence templates deal with only one activity, it is necessary to have a starting time. Using the example above of visiting an exhibition, if we wanted to know when the event "buy ticket" occurs, we would need to set a start time.

The same concept applies to the choice template. The choice template deals with two activation activities, but there are no restrictions, or relation between the two. Instead, the relation templates deal with the relation between two activities; Delay and Deadline are calculated relative to the occurrences of activation and target activities within the traces.

3.2.1 Common notations and definitions

Before providing the definition of the enriched Declare template, we will define the following notation:

- The set A contains all the activity names.
  Example: in the context of buying a ticket to enter an exhibition, A={"buy the ticket", "enter the exhibition"}.
- The set N contains all the timestamps.
  Example: considering activity "buy the ticket" happened at time instant 10, and activity "enter the exhibition" happened at time instant 12, N={10, 12}.
- The execution of an activity is represented through an event in the form of

  e(activityName, timestamp)

  or

  (activityName, timestamp)

  where the activityName ∈ A represent the unique name of an activity, and timestamp ∈ N is the time instant relative to the execution of the activity.

Depending on the adopted metric reference, the timestamp could be, for example the number of milliseconds elapsed since the 1st of January 1970. Each form is used interchangeably throughout this work.

- The execution of any activity is represented with Any

  e(Any, timestamp)
- The trace T is a set of event executions. For example, the trace:

  T={(a,34), (b, 38), (c, 56)}

  represents the execution of the activities a, b, c respectively at time 34, 38 and 56. The timestamps in the events induce a total ascending order between the events belonging to a trace.
- Let T_s (start time) be the timestamp associated with the first event in trace

  $T_s \in N$ such that $\forall(t \in N \text{ and } T_s \neq t) \rightarrow T_s < t$.
- Let T_e (end time) be the timestamp associated with the last event in the trace

  $T_e \in N$ such that $\forall(t \in N \text{ and } T_e \neq t) \rightarrow T_e > t$.
- Delay$\in \mathbb{Z}^+$ and Deadline$\in \mathbb{Z}^+$[7]

3.2.2 Existence templates

The existence templates deal with the presence of an occurrence or multiple occurrences of an activity. Delay and Deadline in general represent the time that passes between the actual occurrence of an activity and the start time.

The *init(A, Delay, Deadline)* constraint template states that the first element of the trace has to start with activity A. In addition to the original template, Delay and

[7] $\mathbb{Z}^+$ is the set of all positive integers.

Deadline are introduced to ensure uniformity in the syntax and will assume no semantical value.

The *last(A, Delay, Deadline)* constraint template states that the last element of the trace has to end with activity A. In addition to the original template, Delay and Deadline are introduced to ensure uniformity in the syntax and will assume no semantical value.

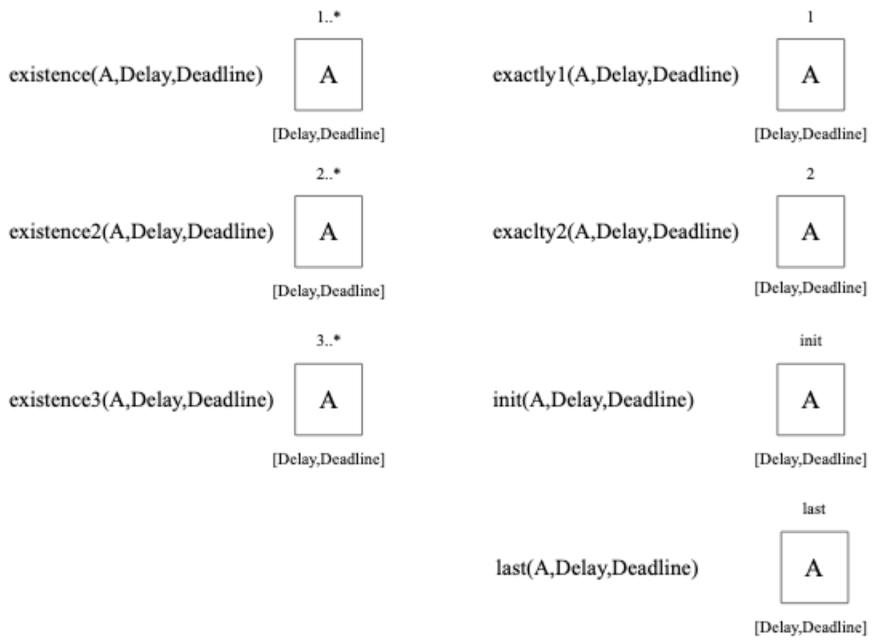

Figure 3.1: Graphical representation for the existence templates enriched with time quantitative metrics

The *existence(A,Delay,Deadline)* constraint template states that activity A has to occur at least once in the trace. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass after the start time, before an activity A is expected to occur. The Deadline variable represents the maximum amount of time that can pass, after the start time, before at least one activity A is expected to occur. At least one activity A has to happen between the time interval delimited by Delay and Deadline.

The semantics of *existence(A,Delay,Deadline)* is as follows:

> Either the trace starts with the event e(A, T_a): in this case the timestamp T_a coincides with the start timestamp T_s
> Or the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that
>
> $$T_a \geq T_s + \text{Delay} \wedge T_a \leq T_s + \text{Deadline}$$

Example 3.1: Considering the constraint *existence(A,3,5)*, trace [(A,1), (C,2), (B,8), (E,10)] satisfies the constraint. On the other hand, the trace [(S,1), (C,2), (A,8), (E,17)] does not satisfy the constraint because no activities with name A happen within the specified time interval.

Example 3.2: Considering the constraint *existence(A,3,6)*, the trace [(S,1), (A,4), (C,8), (E,10)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (C,3), (A,8), (E,17)] does not satisfy the constraint because no activities with name A happen within the specified time interval.

The *existence2(A,Delay,Deadline)* constraint template states that activity A has to occur at least twice in the trace. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the start time, before an activity A is expected to occur. The Deadline variable represents the maximum amount of time that can pass, after the start time, before at least two activities A are expected to occur. At least two activities A have to happen between the time interval delimited by Delay and Deadline.

The semantics of *existence2(A,Delay,Deadline)* is as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $|TA| \geq 2$
>
> Either the trace starts with the event $e(A, T_{a1})$, the timestamp T_{a1} coincides with the start timestamp T_s, and there must be at least one more T_{aj} with $1<j\leq n$ such that
>
> $$T_{aj} \geq T_s+Delay \wedge T_{aj} \leq T_s+Deadline$$
>
> Or the trace starts with an event $e(S, T_s)$ with activity name S different from A, and it holds that $\exists\ T_{a1}, T_{a2}$ such that
>
> $$T_{a1} \geq T_s+Delay \wedge T_{a1} \leq T_s+Deadline$$
> $$T_{a2} \geq T_s+Delay \wedge T_{a2} \leq T_s+Deadline$$

Example 3.3: Considering the constraint: *existence2(A,1,10)*, trace [(S,1), (A,3), (A,8), (E,10)] satisfies the constraint. On the other hand, the trace [(S,1), (C,2), (A,8), (A,17)] does not satisfy the constraint because only one activity with name A happen within the specified time interval.

The *existence3(A,Delay,Deadline)* constraint template states that activity A has to occur at least three times in the trace. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the start time, before an activity A is expected to occur. The Deadline variable represents the maximum amount of time that can pass, after the start time, before at least three activities A are expected to occur. At least three activities A have to happen between the time interval delimited by Delay and Deadline.

The semantics of *existence3(A,Delay,Deadline)* is as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\}\subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an})\in T_a$ and $|TA| \geq 3$

If the trace starts with the event $e(A, T_{a1})$, the timestamp T_{a1} coincides with the start timestamp T_s, and there must be at least two more T_{aj} and T_{ak} with $1<j\leq n$, $1<k\leq n$ and $j\neq k$ such that

$$T_{aj}\geq T_s+Delay \wedge T_{aj}\leq T_s+Deadline$$

$$T_{ak}\geq T_s+Delay \wedge T_{ak}\leq T_s+Deadline$$

Or the trace starts with an event $e(S, T_s)$ with activity name S different from A, and it holds that $\exists$ T_{a1}, T_{a2}, T_{a3} such that

$$T_{a1}\geq T_s+Delay \wedge T_{a1}\leq T_s+Deadline$$

$$T_{a2}\geq T_s+Delay \wedge T_{a2}\leq T_s+Deadline$$

$$T_{a3}\geq T_s+Delay \wedge T_{a3}\leq T_s+Deadline$$

Example 3.4: Considering the constraint: *existence3(A,1,10)*, the trace [(S,1), (A,2), (A,8), (A,10), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (A,8), (A,17)] does not satisfy the constraint because only two activities with name A happen within the specified time interval.

The *exactly1(A,Delay,Deadline)* constraint template states that activity A has to occur exactly one time in the trace. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the start time, before an activity A is expected to occur. The Deadline variable represents the maximum amount of time that can pass, after the start time, before exactly one activities A is expected to occur. Exactly one activity A have to happen between the time interval delimited by Delay and Deadline.

The semantics of *exactly1(A,Delay,Deadline)* is as follows:

> Either the trace starts with the event e(A,T_a): in this case the timestamp T_a coincides with the start timestamp T_s
> Or the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that Given $TA=\{(A,T_a)\}\subseteq T$, $(T_a)\in T_a$ and $|TA| = 1$ such that
> $$T_a \geq T_s + Delay \land T_a \leq T_s + Deadline$$

Example 3.5: Considering the constraint: *exactly1(A,3,10)*, the trace [(S,1), (A,5), (C,8), (A,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (A,8), (E,17)] does not satisfy the constraint because more than one activity with name A happen within the specified time interval.

The *exactly2(A,Delay,Deadline)* constraint template states that activity A has to occur exactly twice in the trace. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the start time, before an activity A is expected to occur. The Deadline variable represents the maximum amount of time that can pass, after the start time, before two activities A are expected to occur. Exactly two activities A have to happen between the time interval delimited by Delay and Deadline.

The semantics of *exactly2(A,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2})\} \subseteq T$, $(T_{a1}, T_{a2}) \in T_a$ and $|TA| = 2$

Either the trace starts with the event $e(A,T_a)$: in this case the timestamp T_a coincides with the start timestamp T_{a2}, and it holds that

$$T_{a2} \geq T_s + Delay \wedge T_{a2} \leq T_s + Deadline$$

Or the trace starts with an event $e(S, T_s)$ with activity name S different from A, and it holds that

$$T_{a1} \geq T_s + Delay \wedge T_{a1} \leq T_s + Deadline$$
$$T_{a2} \geq T_s + Delay \wedge T_{a2} \leq T_s + Deadline$$

Example 3.6: Considering the constraint *exactly2(A, 3, 10)*, the trace [(S,1), (A,5), (A,8), (E,11)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (A,8), (A,9), (E,13)] does not satisfy the constraint because more than two activities with name A happen within the specified time interval.

Table 6 lists existence constraint templates enriched with quantitative time metrics.

Template	**Template with quantitative time**
init(A)	*init(A,Delay,Deadline)* Requires that activity A has to occur as first element of the trace. Delay and Deadline have no semantic value.
last(A)	*last(A,Delay,Deadline)* Requires that activity A has to occur as last element of the trace. Delay and Deadline have no semantic value.

existence(A)	*existence(A,Delay,Deadline)* Requires that the cardinality of activity A has to be at least one, and the timestamp associated to the occurrence of activity A has to be between start time + Delay and start time + Deadline.
existence2(A)	*existence2(A,Delay,Deadline)* Requires that the cardinality of activity A has to be at least two, and the timestamps associated to the occurrences of activity A have to be between start time + Delay and start time + Deadline.
existence3(A)	*existence3(A,Delay,Deadline)* Requires that the cardinality of activity A has to be at least three, and the timestamps associated to the occurrences of activity A have to be between start time + Delay and start time + Deadline.
exaclty1(A)	*exaclty1(A,Delay,Deadline)* Requires that the cardinality of activity A has to be one, and the timestamp associated to the occurrence of activity A has to be between start time + Delay and start time + Deadline.

exaclty2(A)	*exaclty2(A,Delay,Deadline)* Requires that the cardinality of activity A has to be two, and the timestamps associated to the two occurrences of activity A have to be between start time + Delay and start time + Deadline.

Table 6: Existence constraint templates enriched with quantitative time metrics.

3.2.3 Choice templates

The choice templates deal with the presence and absence of activities, they require one activity to be present. The *choice(A,B)* template deals with the existence of at least one of the two activities, but it does not specify any constrains in terms of absence of the other activity. Therefore, although the constraint is verified in a specific interval of time, it remains an assertion made using existential quantification. The *exclusive_choice(A,B)* asserts that one of the two activity is present, but also given the presence of the first activity then the second one is absent or vice versa. The absence of the first or second activity has to be verified universally, across the entire trace, regardless of the time interval taken into consideration. The challenge of defining quantitative temporal variables for constraints dealing with the absence of activities is not in scope for this work. Therefore, only the *choice(A,B)* has been extended to contain quantitative time metrics.

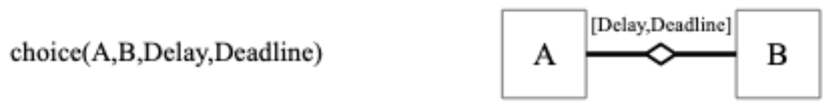

Figure 3.2 Graphical representation for the choice template enriched with time quantitative metrics

The *choice(A,B,Delay,Deadline)* constraint template states that activity A or activity B have to eventually be executed. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the start time, before either an activity A, or an activity B occurs. The Deadline variable represents the maximum amount of time that can pass, after the start time, before either an activity A, or an activity B occurs.

The semantics of *choice(A,B,Delay,Deadline)* are as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\}\subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an})\in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bn})\}\subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bn})\in T_b$, the constraint is satisfied:
>
> If the trace starts with the event $e(A, T_{a1})$ the timestamp T_{a1} coincides with the start timestamp T_s
>
> Or if trace starts with the event $e(B, T_{b1})$ the timestamp T_{b1} coincides with the start timestamp T_s
>
> Or if the trace starts with an event $e(S, T_s)$ with activity name S different from A or B, $\exists t_a\in T_a$, and it holds that
>
> $t_a\geq T_s+Delay \wedge t_a\leq T_s+Deadline$
>
> Or if the trace starts with an event $e(S, T_s)$ with activity name S different from A or B, $\exists t_b\in T_b$, and it holds that
>
> $$t_b\geq T_s+Delay \wedge t_b\leq T_s+Deadline$$

Example 3.7: Considering the *constraint choice(A,B,3,9)*, the traces [(S,1), (A,5), (A,8), (E,11)], [(S,1), (C,5), (B,7), (E,11)] and [(S,1), (B,6), (A,7), (E,11)] satisfy the constraint. On the other hand, the trace [(S,1), (A,2), (C,8), (C,9), (E,13)] does not satisfy the constraint because neither the activity A nor the activity B happen within the specified time interval.

Table 8 lists choice constraint templates enriched with quantitative time metrics.

Template	Template with quantitative time
choice(A,B)	*choice(A,B,Delay,Deadline)* Requires that activity A or B will eventually occur, and the timestamps associated to either the occurrence of activities A or B have to be between start time + Delay and start time + Deadline.

Table 7: Choice constraint templates enriched with quantitative time metrics.

3.2.4 Relation templates

The positive relation templates deal with the presence of two activities, the activation and target activity. Delay and Deadline represent the minimum and maximum time that passes between the timestamps associated with the activation and the target activities.

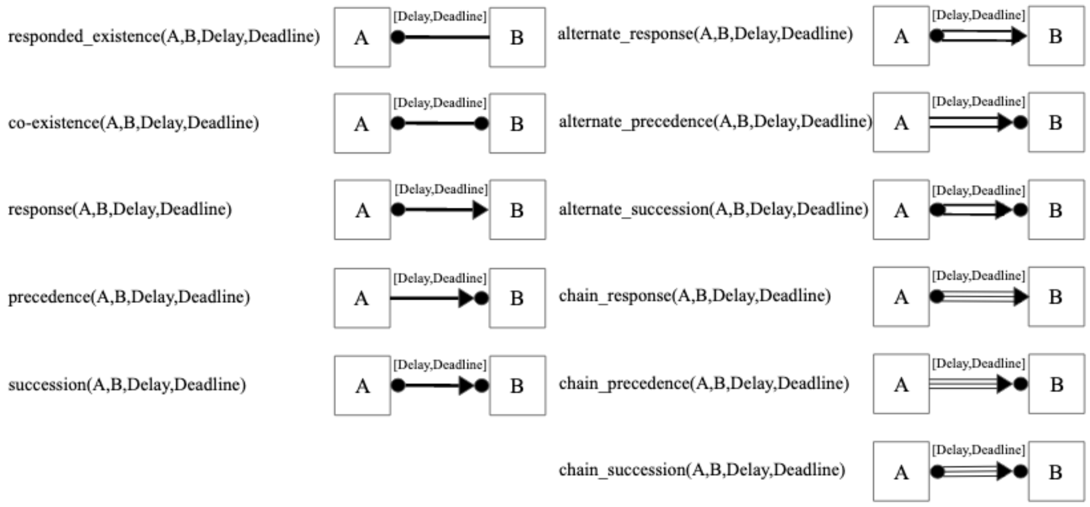
responded_existence(A,B,Delay,Deadline)
co-existence(A,B,Delay,Deadline)
response(A,B,Delay,Deadline)
precedence(A,B,Delay,Deadline)
succession(A,B,Delay,Deadline)
alternate_response(A,B,Delay,Deadline)
alternate_precedence(A,B,Delay,Deadline)
alternate_succession(A,B,Delay,Deadline)
chain_response(A,B,Delay,Deadline)
chain_precedence(A,B,Delay,Deadline)
chain_succession(A,B,Delay,Deadline)
A
B
[Delay,Deadline]

Figure 3.3: Graphical representation for the relation templates enriched with time quantitative metrics

The *responded_existence(A,B,Delay,Deadline)* template states that if activity A is executed then activity B has to be executed either before or after activity A. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, or has passed, from the occurrence of activity A, for the activity B to occur, or have occurred. The Deadline variable represents the maximum amount of time that can pass, or should have passed, from the occurrence of the activity A, for the activity B to occur, or should have occurred.

The semantics of *responded_existence(A,B,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a\ \exists t_{bj} \in T_b$ such that

$$abs(t_{bj}-t_{ai}) \geq Delay \land abs(t_{bj}-t_{ai}) \leq Deadline$$

Example 3.8: Considering the constraint *responded_existence(A,B,2,6)*, the trace [(S,1), (B,2), (A,4), (A,5), (B,10)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (C,8), (B,11), (E,13)] does not satisfy the constraint because the activity B, in relation to the activity A, happens after the specified time interval.

The *co-existence(A,B,Delay,Deadline)* template states that if activity A is executed then activity B is executed and if activity B is executed then activity A is executed. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, or has passed, from the occurrence of the activity A/B, for the activity B/A to occur. The Deadline variable represents the maximum amount of time that can pass, or should have passed, from the occurrence of the activity A/B, for the activity B/A to occur.

The semantics of *co-existence(A,B,Delay,Deadline)* are as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a \ \exists t_{bj} \in T_b$ and $\forall t_{bj} \in T_b \ \exists t_{ai} \in T_a$ such that
>
> $$abs(t_{bj}-t_{ai}) \geq Delay \wedge abs(t_{bj}-t_{ai}) \leq Deadline$$

Example 3.9: Considering the constraint *co-existence(A,B,1,6)*, the trace [(S,1), (C,2), (A,5), (A,9), (B,10)] satisfies the constraint. On the other hand, the trace [(S,1), (B,2), (A,13), (E,14)] does not satisfy the constraint because the activity A, in relation to the activity B, happens after the specified time interval.

The *response(A,B,Delay,Deadline)* template states that whenever activity A is executed, activity B has to eventually be executed after it. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after the activity A has occurred, for the activity B to occur. The Deadline variable represents the maximum amount of time that can pass, after the activity A has occurred, for the activity B to occur.

The semantics of *response(A,B,Delay,Deadline)* are as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a \ \exists t_{bj} \in T_b \ t_{bj} > t_{ai}$ such that
>
> $$t_{bj}-t_{ai} \geq Delay \wedge t_{bj}-t_{ai} \leq Deadline$$

Example 3.10: Considering the constraint *response(A,B,3,5)*, the trace [(S,1), (A,2), (A,4), (B,7), (E,11)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (A,5), (A,6), (B,10)] does not satisfy the constraint because the time that passes between the first execution of activity A, and activity B, is higher than the specified Deadline value.

The *precedence(A,B,Delay,Deadline)* template states that whenever activity B is executed, activity A has to be executed at some point before it. In addition to the original template, the Delay variable represents the minimum amount of time that should have passed, before activity B can occur, from the occurrence of activity A. The Deadline variable represents the maximum amount of time that could have passed, before activity B has to occur, from the occurrence of activity A.

The semantics of *precedence(A,B,Delay,Deadline)* are as follows:

Given TA=$\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and TB=$\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{bj} \in T_b\ \exists t_{ai} \in T_a\ t_{bj} > t_{ai}$ such that

$$t_{bj} - t_{ai} \geq Delay \land t_{bj} - t_{ai} \leq Deadline$$

Example 3.11: Considering the constraint *precedence(A,B,3,6)*, the trace [(S,1), (A,2), (B,5), (B,6), (E,11)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (C,5), (B,10), (E,11)] does not satisfy the constraint because there is no activity A executed, within the specified time interval, before the occurrence of activity B,

The *succession(A,B,Delay,Deadline)* is defined as a combination of the *response(A,B,Delay,Deadline)* constraint and the *precedence(A,B,Delay,Deadline)* one, both constraints have to be verified for the succession constraint to be verified. In addition to the original template, the Delay variable expresses the least amount of time that has to pass between an occurrence of activity A and the occurrence of activity B. The Deadline variable expresses the most amount of time that can pass between an occurrence of activity A and the occurrence of activity B.

The semantics of *succession(A,B,Delay,Deadline)* are as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a\ \exists t_{bj} \in T_b$ and $\forall t_{bj} \in T_b\ \exists t_{ai} \in T_a\ t_{bj} > t_{ai}$ such that
>
> $$t_{bj} - t_{ai} \geq Delay \wedge t_{bj} - t_{ai} \leq Deadline$$

Example 3.12: Considering the constraint *succession(A,B,2,6)*, the trace [(S,1), (A,2), (A,5), (B,7), (B,8), (E,11)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (C,5), (B,9), (E,11)] does not satisfy the constraint because there is no activity B executed, in relation to the occurrence of activity A, within the specified time interval. In this case the response(A,B,2,6) constraint is not verified leading to the trace not being satisfied.

The *alternate_response(A,B,Delay,Deadline)* template states that whenever activity A is executed, activity B has to eventually be executed, and between the executions of two activities A, at least one activity B must be executed. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after activity A has occurred, for activity B to occur. The Deadline variable represents the maximum amount of time that can pass, after activity A has occurred, for activity B to occur.

The semantics of *alternate_response(A,B,Delay,Deadline)* are as follows:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a\ \exists t_{bj} \in T_b$, $t_{ai} < t_{bj}$ such that
>
> $$t_{bj} - t_{ai} \geq Delay \wedge t_{bj} - t_{ai} \leq Deadline$$
>
> and $\nexists t_{a(i+1)} \in T_a$ with $t_{ai} < t_{a(i+1)} < t_{bj}$

Example 3.13: Considering the constraint *alternate_response(A,B,3,6)*, the trace [(B,1), (A,2), (C,4), (B,5), (A,7), (B,11), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (B,2), (A,5), (C,6), (B,13), (E,15)] does not satisfy the constraint because there is no activity B executed, in relation to the occurrence of activity A, within the specified time interval. There are no occurrences of activity B in the time interval [8, 11].

The *alternate_precedence(A,B,Delay,Deadline)* template states that whenever activity B is executed, activity A has to be executed before it, and between the executions of two activities B, at least one activity A must be executed. In addition to the original template, the Delay variable represents the minimum amount of time that should have passed, before activity B can occur, from the occurrence of activity A. The Deadline variable represents the maximum amount of time that could have passed, before activity B has to occur, from the occurrence of activity A.

The semantics of *alternate_precedence(A,B,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\}\subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an})\in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\}\subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm})\in T_b$, it holds that $\forall t_{bj}\in T_b\ \exists t_{ai}\in T_a\ t_{bj}>t_{ai}$ such that

$$t_{bj}-t_{ai}\geq Delay \land t_{bj}-t_{ai}\leq Deadline$$

and $\nexists t_{a(i+1)}\in T_a$ with $t_{ai}<t_{a(i+1)}<t_{bj}$

Example 3.14: Considering the constraint *alternate_precedence(A,B,2,7)*, the trace [(S,1), (A,2), (B,5), (A,5), (B,11), (A,12), (E,13)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (B,3), (A,5), (B,10), (E,11)] does not satisfy the constraint because there is no activity A executed, in relation to the occurrence of the first occurrence of activity B, within the specified time interval.

The *alternate_succession(A,B,Delay,Deadline)* is defined as a combination of the *alternate_response(A,B,Delay,Deadline)* and *alternate_precedence(A,B,Delay,* *Deadline)*, both constraints have to be verified for the alternate succession to be verified. In addition to the original template, the Delay variable expresses the least amount of time that has to pass between an occurrence of activity A and the occurrence of activity B. The Deadline variable expresses the most amount of time that can pass between occurrence of an activity A and the occurrence of activity B.

The semantics of *alternate_ succession(A,B,Delay,Deadline)* are as follows:

> Given TA=$\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$
> and TB=$\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a\ \exists t_{bj} \in T_b$ and $\forall t_{bj} \in T_b\ \exists t_{ai} \in T_a\ t_{bj} > t_{ai}$ such that
> $$t_{bj} - t_{ai} \geq Delay \land t_{bj} - t_{ai} \leq Deadline$$
> and $\nexists t_{a(i+1)} \in T_a$ with $t_{ai} < t_{a(i+1)} < t_{bj}$
> and $\nexists t_{b(j+1)} \in T_b$ with $t_{bi} < t_{b(j+1)} < t_{a(i+1)}$

Example 3.15: Considering the constraint *alternate_succession(A,B,2,9)*, the trace [(S,1), (A,2), (C,3), (B,5), (A,7), (B,9), (A,11), (B,18), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (B,3), (A,5), (B,6), (E,11)] does not satisfy the constraint because for both occurrences of activity A, there is no activity B executed within the specified time intervals.

The *chain_response(A,B,Delay,Deadline)* constraint template states that whenever activity A is executed, activity B has to be executed immediately after it. In addition to the original template, the Delay variable represents the minimum amount of time that has to pass, after activity A has occurred, for activity B to occur. The Deadline variable represents the maximum amount of time that can pass, after the activity A has occurred, for the activity B to occur.

The semantics of *chain_response(A,B,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, and it holds that $\forall t_{ai} \in T_a\ \exists t_{bj} \in T_b$

$$t_{bj} - t_{ai} \geq Delay \land t_{bj} - t_{ai} \leq Deadline$$

and $\nexists (Any, T_c) \in T$ with $t_{ai} < t_c < t_{bj}$ and $Any \neq B$

Example 3.16: Considering the constraint *chain_response(A,B,2,5)*, the trace [(S,1), (A,2), (B,5), (A,7), (B,9), (B,10), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (B,8), (E,11)] does not satisfy the constraint because for both occurrences of activity A, there is no activity B executed within the specified time intervals.

The *chain_precedence(A,B,Delay,Deadline)* constraint template states that whenever activity B is executed, activity B has to be executed immediately before it. In addition to the original template, the Delay variable represents the minimum amount of time that should have passed, before activity B can occur, from the occurrence of activity A. The Deadline variable represents the maximum amount of time that could have passed, before activity B has to occur, from the occurrence of activity A.

The semantics of *chain_precedence(A,B,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, and it holds that $\forall t_{bj} \in T_b\ \exists t_{ai} \in T_a$ such that

$$t_{bj} - t_{ai} \geq Delay \land t_{bj} - t_{ai} \leq Deadline$$

and $\nexists (Any, T_c) \in T$ with $t_{ai} < t_c < t_{bj}$ and $Any \neq B$

Example 3.17: Considering the constraint *chain_precedence(A,B,1,4)*, the trace [(S,1), (A,2), (B,5), (A,7), (B,8), (A,10), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (B,8), (A,9), (E,11)] does not satisfy the constraint because for both occurrences of activity A, there is no activity B executed within the specified time intervals.

The *chain_succession(A,B,Delay,Deadline)* is defined as a combination of the template *chain_response(A,B,Delay,Deadline)* and the template *chain_precedence(A ,B,Delay,Deadline)*, both constraints have to be verified for the alternate succession to be verified. In addition to the original template, the Delay variable expresses the least amount of time that has to pass between an occurrence of activity A and the occurrence of activity B. The Deadline variable expresses the maximum amount of time that can pass between an occurrence of activity A and the occurrence of activity B.

The semantics of *chain_ succession(A,B,Delay,Deadline)* are as follows:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, and it holds that $\forall t_{ai} \in T_a$ $\forall t_{bj} \in T_b$ and $\forall t_{bj} \in T_b$ $\exists t_{ai} \in T_a$ such that

$$t_{bj} - t_{ai} \geq Delay \land t_{bj} - t_{ai} \leq Deadline$$

and $\nexists (Any, T_c) \in T$ with $t_{ai} < t_c < t_{bj}$ and $Any \neq B$

Example 3.18: Considering the constraint: *chain_precedence(A,B,1,6)*, the trace [(S,1), (A,2), (B,5), (A,7), (B,8), (E,12)] satisfies the constraint. On the other hand, the trace [(S,1), (A,2), (B,8), (E,11)] does not satisfy the constraint because for both occurrences of activity A, there is no activity B executed within the specified time intervals.

Table 8 lists relation constraint templates enriched with quantitative time metrics.

Template	Template with quantitative time
responded_existence(A,B)	*responded_existence(A,B,Delay,Deadline)* Requires that every time activity A executes, activity B has to be executed either before or after A. The difference, in absolute value, between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
co-existence(A,B)	*co-existence(A,B,Delay,Deadline)* Requires that every time activity A executes, activity B has to be executed as well and vice versa. The difference, in absolute value, between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
response(A,B)	*response(A,B,Delay,Deadline)* Requires that every time activity A executes, activity B has to be executed after it. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
precedence(A,B)	*precedence(A,B,Delay,Deadline)* Requires activity B to be preceded by activity A. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to

	the occurrence of activity B, has to be between Delay and Deadline.
succession(A,B)	*succession(A,B,Delay,Deadline)* Requires that both response and precedence templates have to hold between activities A and B. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
alternate_response(A,B)	*alternate_response(A,B,Delay,Deadline)* Requires that after the execution of activity A, activity B has to be executed and between the execution of each two activities A at least one activity B has to be executed. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
alternate_precedence(A,B)	*alternate_precedence(A,B,Delay,Deadline)* Requires that every instance of activity B has to be preceded by an instance of activity A and next instance of Activity B cannot be executed before the next instance of activity A is executed. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
alternate_succession(A,B)	*alternate_succession(A,B,Delay,Deadline)*

	Requires that both alternate_response and alternate_precedence templates have to hold between activities A and B. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
chain_response(A,B)	*chain_response(A,B,Delay,Deadline)* Requires that next activity after activity A has to be activity B. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
chain_precedence(A,B)	*chain_precedence(A,B,Delay,Deadline)* Requires that the activity A is the first preceding activity before B. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.
chain_succession(A,B)	*chain_succession(A,B,Delay,Deadline)* Requires that both chain_response and chain_precedence templates have to hold between activities A and B. The difference between the timestamp associated to the occurrence of activity A, and the timestamp associated to the occurrence of activity B, has to be between Delay and Deadline.

Table 8: Relation constraint templates enriched with quantitative time metrics

4 Discovering augmented Declare templates

This chapter discusses how the Delay and Deadline variables are calculated for each enriched template and the adaptation made to the existing algorithm, presented in chapter 2, to support the discovery of quantitative time metrics.

The Delay and Deadline variables are introduced in the specialization phase for the DNF mining algorithm and in the generalization phase for the CNF mining algorithm. Both the specialization and generalization phases occur within the *choose_constraint* function. Each enriched constraint is then verified against the positive and negative traces independently. The output of the outer cycle is a model containing the mined quantitative time for each constraint.

4.1 How to determine the constraints

The following outlines how the Delay and Deadline variables are calculated for each enriched template. The templates are discovered over the entire trace.

4.1.1 Existence templates

The variables Delay and Deadline, for the *existence(A,Delay,Deadline)*, are calculated as the difference between the timestamps associated, respectively, with the first and last occurrence of activity A in the trace, and the start time:

If the trace starts with the event e(A, T_{a1}), the timestamp T_{a1} coincides with the start timestamp Ts, and it holds that

$\forall t_{ai} \in T_a \mid t_{ai} > T_{a1}$

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Or the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that $\forall t_{ai} \in T_a$ such that

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Example 4.1: Considering the trace [(S,1), (A,2), (A,5), (B,8), (A,9)] Delay will assume value of 1, while Deadline will assume value of 8.

The variables Delay and Deadline, for the *existence2(A,Delay,Deadline)*, are calculated as the difference between the timestamps associated, respectively, with the first and last occurrence of activity A in the trace, and the start time:

Given TA={(A,T_{a1}), (A,T_{a2}), …, (A,T_{an})}$\subseteq$T, (T_{a1}, T_{a2}, …, T_{an})$\in T_a$ and $|TA| \geq 2$

If the trace starts with the event e(A, T_{a1}), the timestamp T_{a1} coincides with the start timestamp Ts, and it holds that

$\forall t_{ai} \in T_a | t > T_{a1}$

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Or the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that $\forall t_{ai} \in T_a$ such that

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Example 4.2: Considering the trace [(A,1), (A,2), (B,5), (A,8), (C,9)] Delay will assume value 1, while Deadline will assume value 7.

Example 4.3: Considering the trace [(S,1), (A,2), (B,5), (C,8), (A,9)] Delay will assume value of 1, while Deadline will assume value of 8. If there are only two occurrences of activity A, then Delay will represent the difference between the timestamp associated with the first occurrence of activity A and the start time. Deadline will represent the difference between the timestamp associated with the second occurrence of activity A and the start time.

The variables Delay and Deadline, for the *existence3(A,Delay,Deadline)*, are calculated as the difference between the timestamps associated, respectively, with the first and last occurrence of activity A in the trace, and the start time:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $|TA| \geq 3$

If the trace starts with the event $e(A, T_{a1})$, the timestamp T_{a1} coincides with the start timestamp T_s, and it holds that

$\forall t_{ai} \in T_a | t > T_{a1}$

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Or the trace starts with an event $e(S, T_s)$ with activity name S different from A, and it holds that $\forall t_{ai} \in T_a$ such that

Delay = arg min($t_{ai} - T_s$)

Deadline = arg max($t_{ai} - T_s$)

Example 4.4: Considering the trace [(C,1), (A,2), (A,8), (A,10)] Delay will assume value of 1, while Deadline will assume value of 9.

The variables Delay and Deadline, for *exactly1(A,Delay,Deadline)*, are calculated as the difference between the timestamp associated with the only occurrence of activity A in the trace and the start time:

> If the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that Given TA={(A,T_a)}⊆T, (T_a)∈T_a and $|TA| = 1$ such that
>
> Delay= $T_a - T_s$
>
> Deadline= $T_a - T_s$

Example 4.5: Considering the trace [(S,1), (B,3), (A,5), (C,8)] Delay and Deadline will both assume the same value of 4.

The variables Delay and Deadline, for the *exactly2(A,Delay,Deadline)*, are calculated as the difference between the timestamps associated, respectively, with the first and second occurrence of activity A in the trace, and the start time:

> If the trace starts with an event e(S, T_s) with activity name S different from A, and it holds that Given TA={(A,T_{a1}), (A,T_{a2})}⊆T, (T_{a1}, T_{a2})∈T_a and $|TA| = 2$ such that
>
> Delay= $T_{a1} - T_s$
>
> Deadline= $T_{a2} - T_s$

Example 4.6: Considering the trace [(C,1), (A,3), (A,5), (C,8)] Delay will assume value of 2, while Deadline will assume value of 4.

4.1.2 Choice templates

The variables Delay and Deadline, for the *choice(A,B,Delay,Deadline)*, are calculated as the difference between the timestamps associated, respectively, with the first and last occurrence of either activity A or B in the trace, and the start time.

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\}\subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an})\in T_a$,

$TB=\{(B,T_{b1}), (B,T_{b2}), \ldots ,(B,T_{bn})\}\subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bn})\in T_b$

If the trace starts with the event $e(A, T_{a1})$: in this case the timestamp T_{a1} coincides with the start timestamp T_s, and it holds that

$\forall t_{ai}\in T_a \mid t>T_{a1}$

Delay = $\arg\min(t_{ai}-T_s)$

Deadline = $\arg\max(t_{ai}-T_s)$

Or if trace starts with the event $e(B, T_{b1})$: in this case the timestamp T_{b1} coincides with the start timestamp T_s, and it holds that

$\forall t_{bj}\in T_b \mid t>T_{b1}$

Delay = $\arg\min(t_{bj}-T_s)$

Deadline = $\arg\max(t_{bi}-T_s)$

Or if the trace starts with an event $e(S, T_s)$ with activity name S different from A or B, $\forall t_{ai} \in T_a$, and it holds that

Delay = $\arg\min(t_{ai}-T_s)$

Deadline = $\arg\max(t_{ai}-T_s)$

Or if the trace starts with an event $e(S, T_s)$ with activity name S different from A or B, $\forall t_{bj} \in T_b$, and it holds that

Delay = $\arg\min(t_{bj}-T_s)$

Deadline = $\arg\max(t_{bi}-T_s)$

Example 4.7: Considering the trace [(S,1), (A,5), (C,7), (A,19)] Delay will assume value of 4, while Deadline will assume value of 18.

Example 4.8: Considering the trace [(S,1), (B,5), (B,7), (E,19)] Delay will assume value of 4, while Deadline will assume value of 6.

4.1.3 Relation templates

The variables Delay and Deadline, for *responded_existence(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrence of activity A and activity B. Both times are considered with an absolute value:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a$ and $\forall t_{bj} \in T_b$

$$Delay = \arg\min(abs(t_{bj} - t_{ai}))$$

$$Deadline = \arg\max(abs(t_{bj} - t_{ai}))$$

Example 4.9: Considering the trace [(B,1), (A,3), (A,4), (B,9)] Delay represents the difference between the timestamp associated to the first occurrence of activity A and the timestamp associated to the first occurrence of activity B, the absolute value is 2. While Deadline represents the difference between the timestamp associated to the second occurrence of activity A and the timestamp associated to the second occurrence of activity B, the value is 5.

The variables Delay and Deadline, for *co-existence(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrences of activities A and B. Both times are considered with an absolute value:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a$ $\forall t_{bj} \in T_b$

$$Delay = \arg\min(abs(t_{bj} - t_{ai}))$$

$$Deadline = \arg\max(abs(t_{bj} - t_{ai}))$$

Example 4.10: Considering the trace [(B,1), (A,5), (A,10), (B,11)] Delay will assume the value of 1, Deadline will assume the value of 4.

The variables Delay and Deadline, for *response(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrence of any activity A and any subsequent activity B:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \dots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \dots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \dots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \dots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a \ \forall t_{bj} \in T_b \ t_{bj} > t_{ai}$
>
> $$\text{Delay} = \arg\min(t_{bj} - t_{ai})$$
> $$\text{Deadline} = \arg\max(t_{bj} - t_{ai})$$

Example 4.11: Considering the trace [(A,1), (A,2), (A,5) ,(B,8) ,(B,9)] Delay will assume the value of 3, while the Deadline value will assume the value 7.

The variables Delay and Deadline, for *precedence(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrence of any activity A and any subsequent activity B:

> Given $TA=\{(A,T_{a1}), (A,T_{a2}), \dots,(A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \dots,T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \dots,(B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \dots,T_{bm}) \in T_b$, it holds that $\forall t_{bj} \in T_b \ \forall t_{ai} \in T_a \ t_{bj} > t_{ai}$
>
> $$\text{Delay} = \arg\min(t_{bj} - t_{ai})$$
> $$\text{Deadline} = \arg\max(t_{bj} - t_{ai})$$

Example 4.12: Considering the trace [(A,1), (C,5), (B,6), (B,10), (A,11)] Delay will assume the value of 5, while the Deadline value will assume the value of 9.

The variables Delay and Deadline, for *succession(A,B,Delay,Deadline)* are calculated by verifying first *response(A,B,Delay,Deadline)* and then *precedence(A,B,Delay,Deadline)*.

Example 4.13: Considering the trace [(A,1), (C,2), (A,5), (B,7), (B,8)] Delay will assume the value of 2, while the Deadline value will assume the value of 6.

The variables Delay and Deadline, for *alternate_response(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrences of activities A and B:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{ai} \in T_a \; \forall t_{bj} \in T_b, t_{ai} < t_{bj}$

$$Delay = \arg\min(t_{bj} - t_{ai})$$

$$Deadline = \arg\max(t_{bj} - t_{ai})$$

and $\nexists t_{a(i+1)} \in T_a$ with $t_{ai} < t_{a(i+1)} < t_{bj}$

Example 4.14: Considering the trace [(B,1), (A,2), (C,5), (B,6), (B,8), (A,11), (B,18)] Delay will assume the value of 4, while the Deadline value will assume the value of 7.

The variables Delay and Deadline, for *alternate_precedence(A,B,Delay, Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrences of activities A and B:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$ and $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, it holds that $\forall t_{bj} \in T_b \; \forall t_{ai} \in T_a \; t_{bj} > t_{ai}$

$$Delay = \arg\min(t_{bj} - t_{ai})$$

$$Deadline = \arg\max(t_{bj} - t_{ai})$$

and $\nexists t_{a(i+1)} \in T_a$ with $t_{ai} < t_{a(i+1)} < t_{bj}$

Example 4.15: Considering the trace [(A,1), (C,2), (B,5), (A,7), (B,8), (A,11)] Delay will assume the value of 1, while the Deadline value will assume the value of 4.

The variables Delay and Deadline, for *alternate_succession (A,B,Delay,Deadline)* are calculated by verifying first *alternate_response(A,B,Delay,Deadline)* and then *alternate_precedence(A,B,Delay,Deadline)*.
Example 4.16: Considering the trace [(A,1), (C,2), (B,5), (A,7), (B,8), (A,11), (B,18)] Delay will assume the value of 1, while the Deadline value will assume the value of 7.

The variables Delay and Deadline, for *chain_response(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrences of activities A and B:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, and it holds that $\forall t_{ai} \in T_a \; \exists t_{bj} \in T_b$

$$\text{Delay} = \arg\min(t_{bj} - t_{ai})$$
$$\text{Deadline} = \arg\max(t_{bj} - t_{ai})$$

and $\nexists(\text{Any}, T_c) \in T$ with $t_{ai} < t_c < t_{bj}$ and $\text{Any} \neq B$

Example 4.17: Considering the trace [(B,1), (A,2), (B,5), (C,7), (A,8), (B,14)] Delay will assume the value of 3, while the Deadline value will assume the value of 6.

The variables Delay and Deadline, for *chain_precedence(A,B,Delay,Deadline)*, are calculated, respectively, as the minimum and maximum time difference between the occurrences of activities A and B:

Given $TA=\{(A,T_{a1}), (A,T_{a2}), \ldots, (A,T_{an})\} \subseteq T$, $(T_{a1}, T_{a2}, \ldots, T_{an}) \in T_a$, $TB=\{(B,T_{b1}), (B,T_{b2}), \ldots, (B,T_{bm})\} \subseteq T$, $(T_{b1}, T_{b2}, \ldots, T_{bm}) \in T_b$, and it holds that $\forall t_{bj} \in T_b \; \exists t_{ai} \in T_a$

$$\text{Delay} = \arg\min(t_{bj} - t_{ai})$$
$$\text{Deadline} = \arg\max(t_{bj} - t_{ai})$$

and $\nexists(\text{Any}, T_c) \in T$ with $t_{ai} < t_c < t_{bj}$ and $\text{Any} \neq B$

Example 4.18: Considering the trace [(A,1), (B,2), (C,5), (A,7), (B,13), (A,14)] Delay will assume the value of 1, while the Deadline value will assume the value of 6.

The variables Delay and Deadline, for *chain_succession (A,B,Delay,Deadline)* are calculated by verifying first *chain_response(A,B,Delay,Deadline)* and then precedence *chain_precedence(A,B,Delay,Deadline)*.
Example 4.19: Considering the trace [(A,1), (B,3), (C,5), (A,7), (B,12), (A,14), (B,19)] Delay will assume the value of 2, while the Deadline value will assume the value 5.

To summarise, the learning strategy adopted for each template is to discover the greatest and the smallest time intervals within which the constraint is verified over the entire trace. The discovery does not stop at the first occurrence of the template. As a result, knowledge of the logs is general and not specific.
Example 4.20: Let us take in consideration the trace [(A,1), (B,3), (C,5), (A,7), (B,12), (A,14), (B,15)] and the constraint *chain_succession(A,B,Delay,Deadline)*. If a specific learning strategy is adopted, the discovery process can stop after the first time the constraint is verified over the trace. In this case the value of the variables Delay, and Deadline is 2. This may limit the analysis of the business process, restricting it to a specific occurrence of a constraint in a trace. Instead, if a generalised learning strategy is adopted, the discovery process continues over the entire trace, giving a more complete and holistic analysis. In this case the value of the Delay variable is 1 and the value of the Deadline variable is 5.

4.1.4 Leaning the temporal constraints over a finite set of traces

In the current implementation, Delay and Deadline are evaluated using constraint logic programming over a finite domain. This makes it possible to reason and solve complex computation with minimal code.

Each template is verified over every trace in the logs. This verification step happens in the *get_number_of_satisfied_traces* function. The recursive function takes two inputs: the constraint to verify and a list of traces and returns the number of verified traces. Each time a constraint is verified over a trace the CLP constraints are updated.

Let us consider a list of two traces: trace([event(A,1), event(B,2), event(A,5), event(B, 8), event(B, 9)]) and trace([event(A,1), event(B,4), event(A,5), event(B, 20)]). Let us also consider *response(A,B,Delay,Deadline)* as the constraint to verify. The constraint is verified in sequence over the first and second trace. The Delay variable assumes the value of one, and the Deadline variable assumes the value of 15. The Delay value is determined by verifying the constraint over the first trace, while the Deadline value is determined by verifying the constraint over the second trace.

To summarise, using CLP, arithmetic constraints are added to the temporal variables, and every time a template is verified over a trace, more generalised/specific constraints are discovered.

4.2 Enriched hierarchy of templates

The specialization phase for DNF algorithm and the generalization phase for the CNF algorithm are based upon the hierarchy of templates. Therefore, the subsumption maps, as described by Elena Palmieri in [3], are enriched with the two variables Delay and Deadline representing the quantitative time metrics.

Delay and Deadline variables assume different meanings for the different template groups. Delay and Deadline in relation to the existence template groups represent the time between one or more occurrences of the specified activity and the start time, while for the relation templates group, they represent the minimum and the maximum time that passes between the activation and target activities. In the case of templates that deal with the absence of activities, Delay and Deadline are not defined. The different semantics associated with Delay and Deadline for the different groups of templates lead to restricting the inheritance only to templates belonging to the same group.

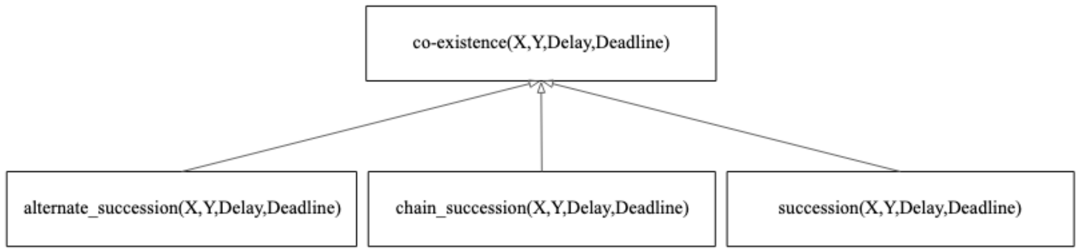
co-existence(X,Y,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)
succession(X,Y,Delay,Deadline)

Figure 4.1: Subsumption map of the *co-existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

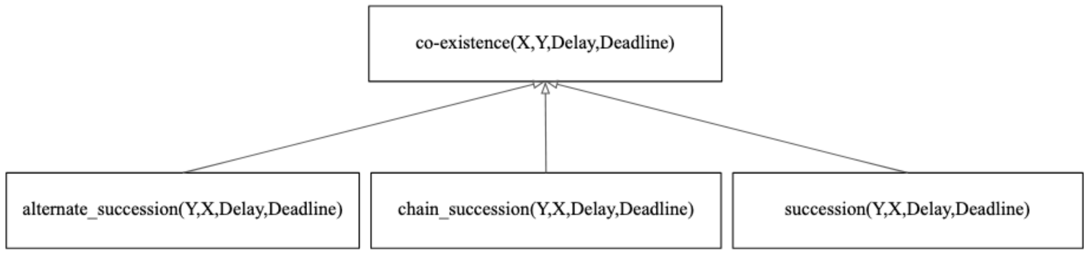
co-existence(X,Y,Delay,Deadline)
alternate_succession(Y,X,Delay,Deadline)
chain_succession(Y,X,Delay,Deadline)
succession(Y,X,Delay,Deadline)

Figure 4.2: Subsumption map of the *co-existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

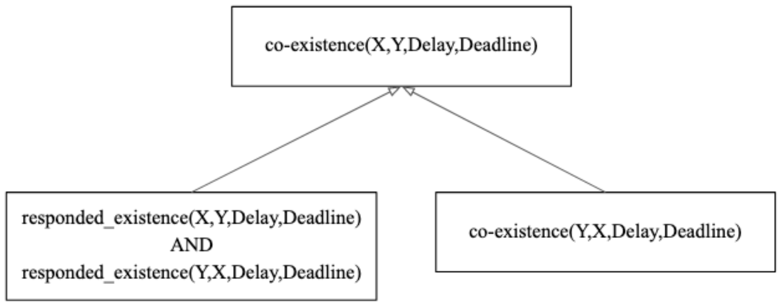
co-existence(X,Y,Delay,Deadline)
responded_existence(X,Y,Delay,Deadline)
AND
responded_existence(Y,X,Delay,Deadline)
co-existence(Y,X,Delay,Deadline)

Figure 4.3: Subsumption map of the *co-existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

Considering the subsumption map for the co-existence template shown in Figure 4.1, Figure 4.2 and Figure 4.3, the *alternate_succession(X,Y,Delay,Deadline)*, *chain_succession(X,Y,Delay, Deadline)*, *succession(X,Y,Delay,Deadline)* and the conjunction of responded_existence*(X,Y,Delay,Deadline)* and responded_existence *(Y,X,Delay,Deadline)*, all strengthen the condition stated by the *co-existence(X,Y,Delay,Deadline)* template. The condition is strengthened by specifying that not only the activation and target activities have to occur but also the relative position between the two activities within the trace. The Delay and Deadline variables are inherited, since the time range that passes between the activities does not change, given the constraints are verified.

The trace T=[(A,1), (B,3), (C,5), (A,7), (B,12), (A,14), (B,19)] satisfies the *co-existence(X,Y,Delay,Deadline)* constraint, and Delay and Deadline variables assume the values 2 and 5 respectively. Taking into consideration the first branch of the *co-existence(X,Y,Delay,Deadline)* subsumption map in Figure 4.1, the *alternate_succession(X,Y,Delay,Deadline)*, *chain_succession(X,Y,Delay,Deadline)* and *succession(X,Y,Delay,Deadline)* are verified over trace T, and Delay and Deadline variables assume the values 2 and 5 respectively for the three specialised constraints.

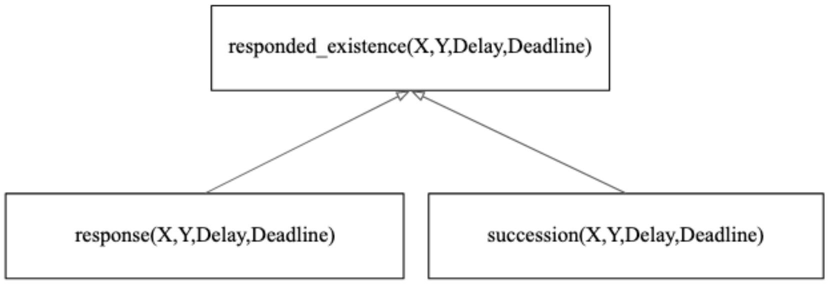
responded_existence(X,Y,Delay,Deadline)
response(X,Y,Delay,Deadline)
succession(X,Y,Delay,Deadline)

Figure 4.4: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

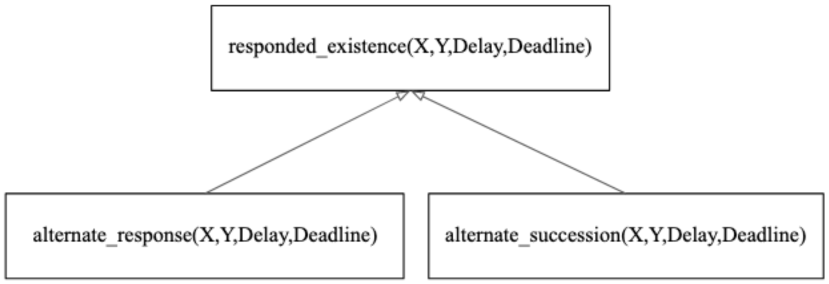
responded_existence(X,Y,Delay,Deadline)
alternate_response(X,Y,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)

Figure 4.5: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

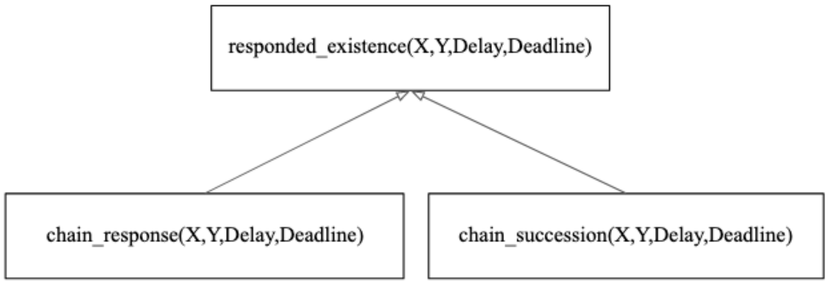
responded_existence(X,Y,Delay,Deadline)
chain_response(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)

Figure 4.6: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

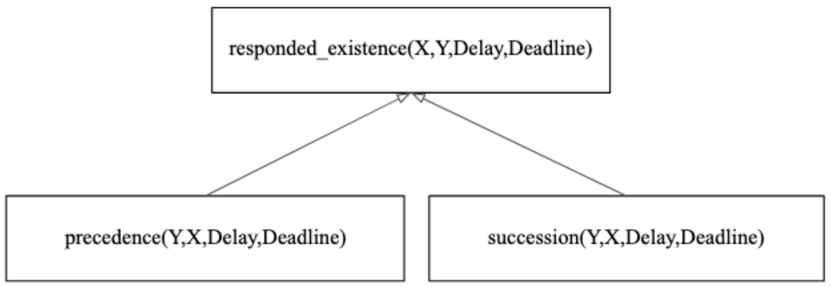
responded_existence(X,Y,Delay,Deadline)
precedence(Y,X,Delay,Deadline)
succession(Y,X,Delay,Deadline)

Figure 4.7: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

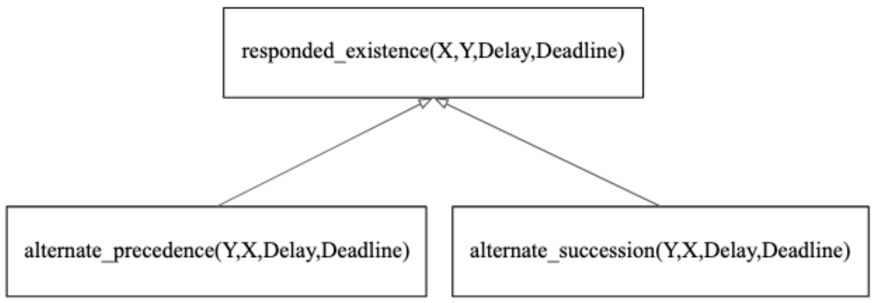
responded_existence(X,Y,Delay,Deadline)
alternate_precedence(Y,X,Delay,Deadline)
alternate_succession(Y,X,Delay,Deadline)

Figure 4.8: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

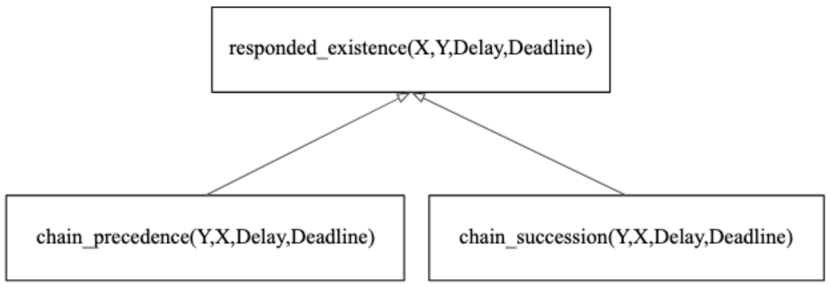
responded_existence(X,Y,Delay,Deadline)
chain_precedence(Y,X,Delay,Deadline)
chain_succession(Y,X,Delay,Deadline)

Figure 4.9: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metrics.

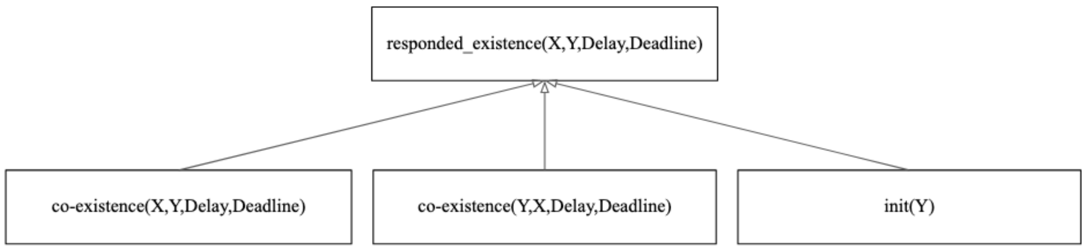
responded_existence(X,Y,Delay,Deadline)
co-existence(X,Y,Delay,Deadline)
co-existence(Y,X,Delay,Deadline)
init(Y)

Figure 4.10: Subsumption map of the *responded_existence(X,Y,Delay,Deadline)* template with quantitative time metric

Considering the *responded_existence(X,Y,Delay,Deadline)* subsumption map as shown in Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, Figure 4.8, Figure 4.9 and Figure 4.10, all the specialised relation constraints inherit Delay and Deadline. This is because they strengthen the condition, relative to the position of the activation and target activities occurrences. Delay and Deadline, for the init template, are added for syntactic completeness, and therefore are not inherited as shown in Figure 4.10.

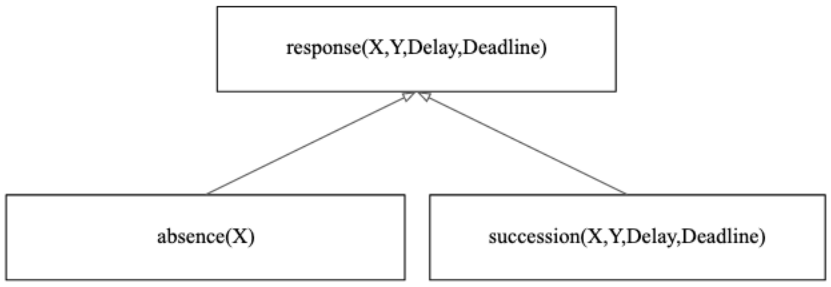

Figure 4.11: Subsumption map of the *response(X,Y,Delay,Deadline)* template with quantitative time metrics.

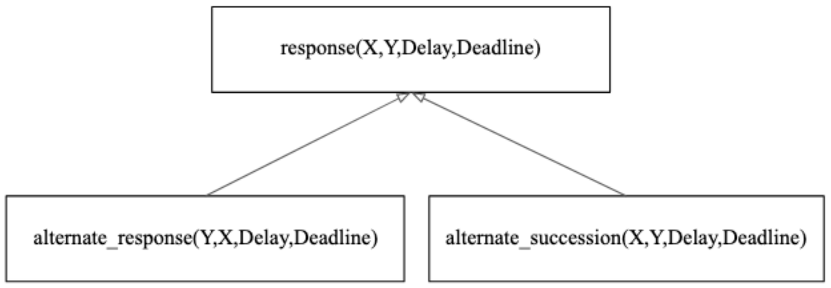
response(X,Y,Delay,Deadline)
alternate_response(Y,X,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)

Figure 4.12: Subsumption map of the *response(X,Y,Delay,Deadline)* template with quantitative time metrics.

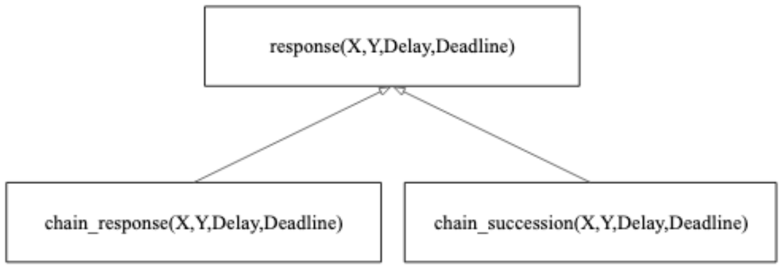
response(X,Y,Delay,Deadline)
chain_response(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)

Figure 4.13: Subsumption map of the *response(X,Y,Delay,Deadline)* template with quantitative time metrics.

In the *response(X,Y,Delay,Deadline)* subsumption map, all the specialised relation constraints inherit Delay and Deadline as shown in Figure 4.11, Figure 4.12 and Figure 4.13. Delay and Deadline are not inherited by the *absence(X)* template as the quantitative time metrics for non-existence templates have not been defined as shown in Figure 4.11.

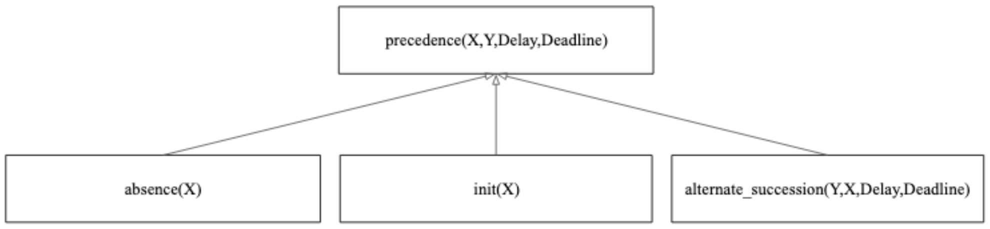
precedence(X,Y,Delay,Deadline)
absence(X)
init(X)
alternate_succession(Y,X,Delay,Deadline)

Figure 4.14: Subsumption map of the *precedence(X,Y,Delay,Deadline)* template with quantitative time metrics.

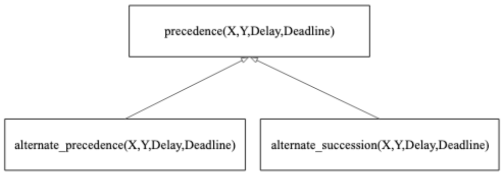
precedence(X,Y,Delay,Deadline)
alternate_precedence(X,Y,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)

Figure 4.15: Subsumption map of the *precedence(X,Y,Delay,Deadline)* template with quantitative time metrics.

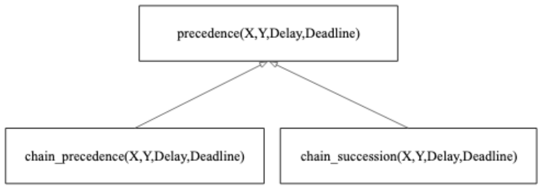
precedence(X,Y,Delay,Deadline)
chain_precedence(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)

Figure 4.16: Subsumption map of the *precedence(X,Y,Delay,Deadline)* template with quantitative time metrics.

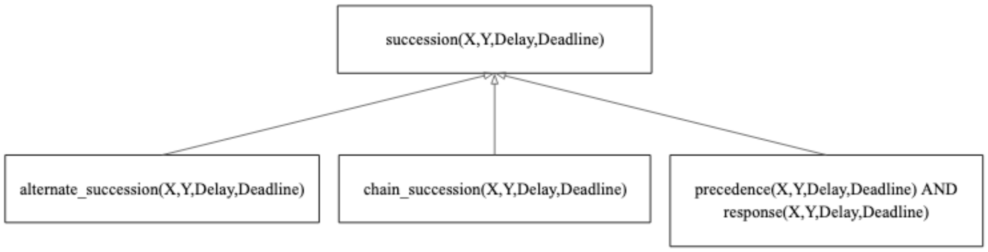
succession(X,Y,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)
precedence(X,Y,Delay,Deadline) AND response(X,Y,Delay,Deadline)

Figure 4.17: Subsumption map of the *succession(X,Y,Delay,Deadline)* template with quantitative time metrics.

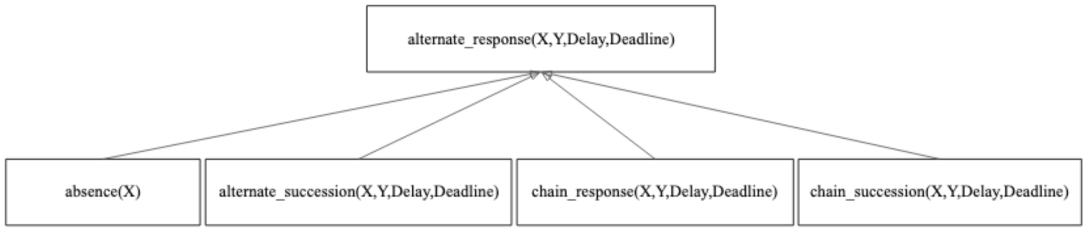
alternate_response(X,Y,Delay,Deadline)
absence(X)
alternate_succession(X,Y,Delay,Deadline)
chain_response(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)

Figure 4.18: Subsumption map of the *alternate_response(X,Y,Delay,Deadline)* template with quantitative time metrics.

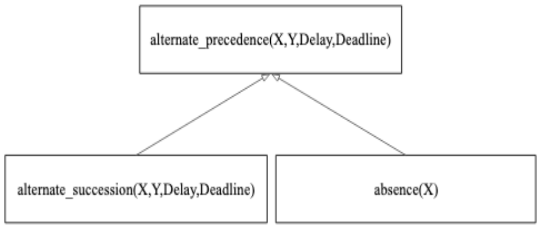
alternate_precedence(X,Y,Delay,Deadline)
alternate_succession(X,Y,Delay,Deadline)
absence(X)

Figure 4.19: Subsumption map of the *alternate_precedence(X,Y,Delay,Deadline)* template with quantitative time metrics.

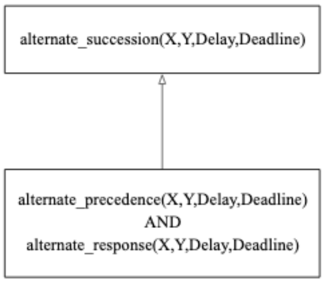
alternate_succession(X,Y,Delay,Deadline)
alternate_precedence(X,Y,Delay,Deadline) AND alternate_response(X,Y,Delay,Deadline)

Figure 4.20: Subsumption map of the *alternate_succession(X,Y,Delay,Deadline)* template with quantitative time metrics.

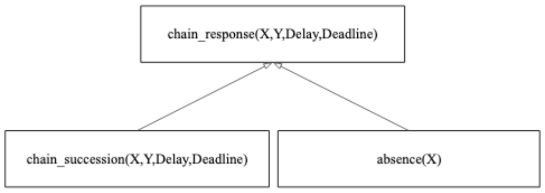
chain_response(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)
absence(X)

Figure 4.21: Subsumption map of the *chain_response(X,Y,Delay,Deadline)* template with quantitative time metrics.

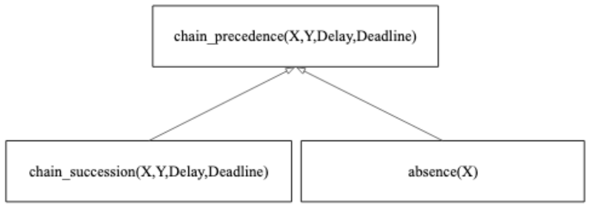
chain_precedence(X,Y,Delay,Deadline)
chain_succession(X,Y,Delay,Deadline)
absence(X)

Figure 4.22: Subsumption map of the *chain_precedence(X,Y,Delay,Deadline)* template with quantitative time metrics.

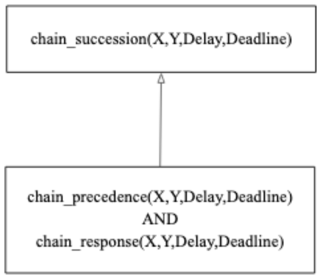
chain_succession(X,Y,Delay,Deadline)
chain_precedence(X,Y,Delay,Deadline) AND chain_response(X,Y,Delay,Deadline)

Figure 4.23: Subsumption map of the *chain_succession(X,Y,Delay,Deadline)* template with quantitative time metrics.

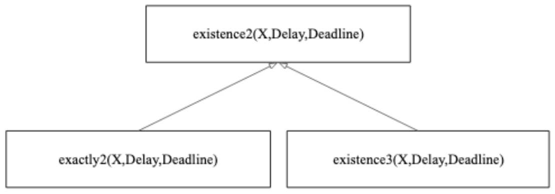
existence2(X,Delay,Deadline)
exactly2(X,Delay,Deadline)
existence3(X,Delay,Deadline)

Figure 4.24: Subsumption map of the *existence2(X,Y,Delay,Deadline)* template with quantitative time metric.

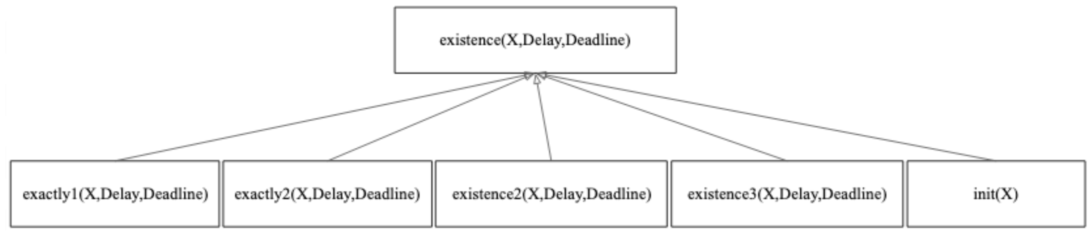
existence(X,Delay,Deadline)
exactly1(X,Delay,Deadline)
exactly2(X,Delay,Deadline)
existence2(X,Delay,Deadline)
existence3(X,Delay,Deadline)
init(X)

Figure 4.25: Subsumption map of the *existence2(X,Y,Delay,Deadline)* template with quantitative time metrics.

None of the specialised constraints in the subsumptions map of the *choice(X,Y,Delay,Deadline)* inherit Delay and Deadline, as the quantitative time variables have different semantics.

The subsumptions maps of the non-existence templates *absence2(X)* and *absence3(X)*; and the subsumptions maps of negation templates *not_chain_succession(X,Y)*, *not_co-existence(X,Y)*, *not_succession(X,Y)*, do not deal with quantitative time metrics as they are not defined. Same applies to the subsumption map of the *exclusive_choice(X,Y)* template.

Currently the algorithm calculates Delay and Deadline when verifying each constraint. This was a necessity to ensure a consistent approach to how the entire set of constraints was verified.

4.3 Disjunctive Normal Form

The specialization phase for the DNF algorithm is executed in the choose_constraint function.

The *choose_constraint* always starts with retrieving the same set of constraints. In the case of the DNF algorithm, this is the first level of hierarchy of the subsumption maps.

The first level of hierarchy is comprised of the constraints: *existence(_)*, *responded_existence(_,_)*, *absence3(_)*, *not_chain_succession(_,_)* and *choice(_,_)*. There are then passed to the *combine* function, which combines each template with every possible combination of activities found in the traces[8]. The variables representing the quantitative time are then added to the first level of hierarchy templates before the specialization phase in the *add_time_to_first_level_of_hierarchy* function.

The output of the *add_time_to_first_level_of_hierarchy* function is the first level of hierarchy template enriched with Delay and Deadline. The enriched constraints are then verified against the positive and negative traces to determine the DNF gain.

Example 4.21: if the set of the activities found in the logs is {a, b, c} then the list of constraint that would be verified is: [responded_existence(a, b, Delay, Deadline), responded_existence(a, c, Delay, Deadline), responded_existence(b, a, Delay, Deadline), responded_existence(b, c, Delay, Deadline), responded_existence(c, a, Delay, Deadline), responded_existence(c, b, Delay, Deadline), existence(a), existence(b), existence(c), absence3(a), absence3(b), absence3(c), choice(a, b, Delay, Deadline), choice(a, c, Delay, Deadline), choice(b, a, Delay, Deadline), choice(b, c, Delay, Deadline), choice(c, a, Delay, Deadline), choice(c, b, Delay, Deadline), not_chain_succession(a, b), not_chain_succession(a, c), not_chain_succession(b, a), not_chain_succession(b, c), not_chain_succession(c, a), not_chain_succession(c, b)]

[8] The constraints are not enriched yet with the two quantitative time variables, this decision has been made to reduce the complexity of the *combine* function.

enhanced DNF choose_constraint function

```
choose_constraint(ListOfPositiveExamples, ListOfNegativeExamples, Term,
  NewConstraint) :-
    get_list_of_activities(ListOfPositiveExamples, ListOfNegativeExamples,
      Activities),
    get_first_level_of_hierarchy(FirstLevelHierarchy)
    combine(FirstLevelOfHierarchy, Activities,
      GroundedFirstLevelOfHierarchy),
    add_time_to_first_level_of_hierarchy(GroundedFirstLevelOfHierarchy,
      GroundedFirstLevelOfHierarchyWithTime),
    specialize_existing_constraints(
      GroundedFirstLevelOfHierarchyWithTime, Term,
      ListOfPossibleCandidates),
    get_best(ListOfPossibleCandidates, ListOfPositiveExamples,
      ListOfNegativeExamples, NewConstraint).
```

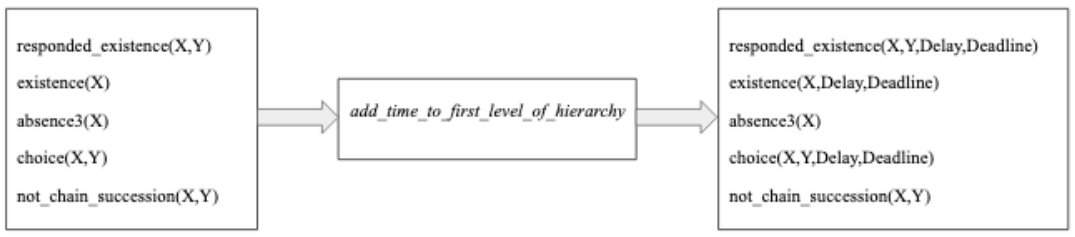
responded_existence(X,Y)
existence(X)
absence3(X)
choice(X,Y)
not_chain_succession(X,Y)
add_time_to_first_level_of_hierarchy
responded_existence(X,Y,Delay,Deadline)
existence(X,Delay,Deadline)
absence3(X)
choice(X,Y,Delay,Deadline)
not_chain_succession(X,Y)

Figure 4.26: *add_time_to_first_level_of_hierarchy* function explained in terms of input and output.

The verified constraints are added in AND to the term. In later iterations the subsumptions maps are used to specialise the constraints in the term.

Example 4.22: if the term contains the constraint [responded_existence(b, c, Delay, Deadline)], then the output of the specialization phase would be: [alternate_response(b, c, Delay, Deadline), alternate_succession(b, c, Delay, Deadline), chain_response(b, c, Delay, Deadline), chain_succession(b, c, Delay, Deadline), co-existence(b, c, Delay, Deadline), response(b, c, Delay, Deadline), succession(b, c, Delay, Deadline), alternate_precedence(c, b, Delay, Deadline), alternate_succession(c, b, Delay, Deadline), chain_precedence(c, b, Delay, Deadline), chain_succession(c, b, Delay, Deadline), co-existence(c, b, Delay, Deadline), init(c, Delay, Deadline), precedence(c, b, Delay, Deadline), succession(c, b, Delay, Deadline)]

The specialised list of constraints is then verified against the positive and negative traces. The final step of the algorithm binds the Delay and Deadline variables associated with each constraint in the list of constraints in AND.

The model containing the quantitative time metrics is then printed to screen.

4.4 Conjunctive Normal Form

The CNF starts from the most specialized set of templates comprised of the constraints: *absence(_)*, *init(_)*, *end(_)*, *exclusive_choice(_,_)*, *existence3(_)*, *exactly2(_)*, *chain_succession(_,_)* and *not_responded_existence(_,_)*; these are then passed to the *combine* function. The variables representing the quantitative time are added to the last level of hierarchy templates before the generalization phase in the *add_time_to_last_level_of_hierarchy* function.

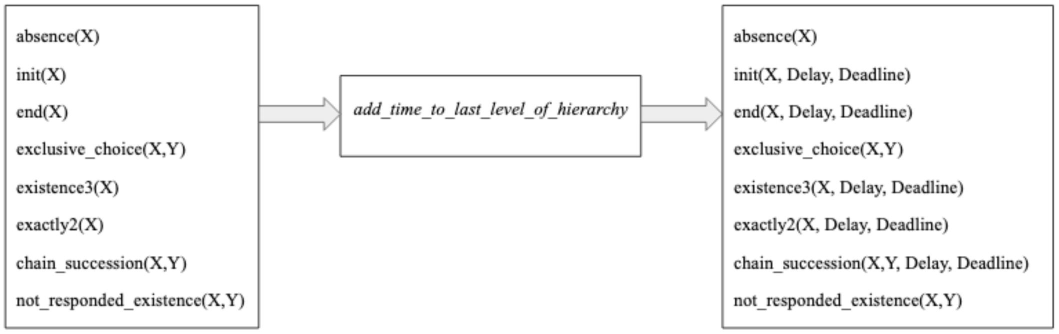
absence(X)
init(X)
end(X)
exclusive_choice(X,Y)
existence3(X)
exactly2(X)
chain_succession(X,Y)
not_responded_existence(X,Y)
add_time_to_last_level_of_hierarchy
absence(X)
init(X, Delay, Deadline)
end(X, Delay, Deadline)
exclusive_choice(X,Y)
existence3(X, Delay, Deadline)
exactly2(X, Delay, Deadline)
chain_succession(X,Y, Delay, Deadline)
not_responded_existence(X,Y)

Figure 4.27: add_time_to_last_level_of_hierarchy function explained in terms of input and output.

Going through the hierarchy in reverse order leads to generalization steps that involve the child nodes having one activity and the parent nodes having two activities. As a result, an extra combination step is needed. The addition of an extra combination step implies that Delay and Deadline variables are removed and then added back before and after the second combination step. The removal of Delay and Deadline does not affect the quality of the mined model as both variables are currently calculated every time a constraint is verified.

4.5 Experimental results

This section reports experimental results on the correctness of the produced process model. The evaluation is performed using a set of controlled event logs[9], relative to a cervical screening, to determine the presence of the papillomavirus infection in women over the age of 25. The logs contain 55 positive traces and 102

[9] The events in the logs are formatted using the eXtendable Event Stream (XES) standard. XES is an XML-based standard for event logs [21].

negative traces. The low number of traces facilitates the verification of the correctness of the process model.

There are 19 activities in the traces:

- execute_biopsy_exam
- execute_colposcopy_exam
- execute_papTest_exam
- send_biopsy_sample
- send_papTest_sample
- send_letter_negative_biopsy
- send_letter_negative_colposcopy
- send_letter_negative_papTest
- send_result_doubt_colposcopy
- send_result_inadequate_papTest
- send_result_negativebiopsy
- send_result_negative_colposcopy
- send_result_negative_papTest
- send_result_positive_biopsy
- send_result_positive_papTest
- invite
- refuse
- phone_call_positive_biopsy
- phone_call_positive_papTest.

The model resulting from the execution of the DNF algorithm is:

```
choice(refuse, send_result_inadequate_papTest,1,3)
OR
(exactly1(send_letter_negative_papTest,4,4)
    AND
    choice(send_letter_negative_papTest,execute_colposcopy_exam,4,11))
```

The reported model states that either the send_result_inadequate_papTest or the refuse activities can occur between one and three units of time after the start time of the process, or that the send_letter_negative_papTest has to occur exactly after 4 units of time after the start time of the process and, either send_letter_negative_papTest or execute_colposcopy_exam can occur within four units and 11 units of time after the start time of the process.

The model, in terms of constraints found, coincided with the model reported in the experimental section in Palmieri’s work [3]. The correctness of reported time intervals was verified against the logs.

5 Conclusion

In this work, an approach to discover declarative process models with quantitative temporal constraint has been presented. The Declare templates are enriched with two variables, Delay and Deadline, with the focus on constraints dealing with the presence of activities. The Delay and Deadline variables are defined for the Declare templates belonging to the existence and relation groups, and the choice template. The two temporal variables for the existence templates and the choice template depend on the definition of the starting point, which is identified as the timestamp associated to the first event in the trace. For the relation templates, the Delay and Deadline variables depend on the timestamps associated to the activation and target events. The activation and target events contain the activation activity and the target activity respectively. The Declare templates dealing with the absence of activity have not been enriched with quantitative temporal variables.

The DNF and CFN algorithms that learn declarative process models from positive and negative traces are modified to return a model containing information about quantitative time measurements. In case of the DNF algorithm, the two variables are introduced during the specialization phase, while for the CNF, they are introduced in the generalization phase. When the model is discovered, it will contain quantitative temporal constraints.

The experimental results, obtained by running the DNF algorithm against the cervical screening logs, have confirmed its correctness. The test was performed to validate the correctness of the model and not the performance of the algorithm. Further testing should be carried out to validate the correctness of the discovered models, using the CNF.

5.1 Future works

5.1.1 Choosing a different start time

The decision to identify a starting point for existence templates and the choice template has been a key point of analysis.

An alternative path to the proposed solution adopted in this work would be the addition of a fictitious event in every trace with a given timestamp. The timestamp needs to be the same for all the traces. Having a common starting time could lead to addressing questions such as “How much time has passed since the opening of the office at 9am to when the first employee has checked in?”. This approach introduces two additional steps before the discovery can be performed. A preliminary analysis of all the traces is needed in order to define the value of such timestamp.

5.1.2 Enrich non-existence Declare templates with quantitative time

Quantifying time, in terms of absences of activities, is an interesting point of discussion and can be explored from different angles. When defining Delay and Deadline, if constraints are checking the presence of activities, we are reasoning over a specific time interval and therefore the model is more specific. In contrast, when dealing with the non-existence of constraints, we are reasoning over the entire trace, hence the model is more generalised.

The *absence(A)* constraint template states that activity A should never been executed in the trace. In the *absence(A,Delay,Deadline)* template, Delay and Deadline could delimit the time interval in which activity A does not have to occur. Delay and Deadline will have to be defined considering a start time.
Let us consider a fraud prevention mechanism that seeks to stop a customer scanning and travelling with a train ticket that they have bought but have requested a refund for.

absence("scan",Delay,Deadline)

The analysis could identify if a customer tries to scan the ticket within the time validity of the ticket itself, given the refund was requested. The Delay variable represents the difference between the start time validity and the purchase time, and the Deadline variable represents the difference between the end time validity and the purchase time

The *absence2(A)* template states that activity A has to occur at most once in the trace. The *absence2(A,Delay,Deadline)* template, Delay and Deadline could delimit the time interval in which the activity A occurs at most once. Delay and Deadline will have to be defined with respect to a defined start time.
Let us consider a fraud prevention mechanism that seeks to stop a customer that has purchased a single train ticket to scan it more than once. The ticket has a time validity associate to it. The customer also has the possibility to have their ticket refunded. This means the customer could scan the ticket at most once within the time validity of the ticket, and zero times in case where the ticket is refunded.

absence2("scan",Delay,Deadline)

The Delay variable represents the difference between the start time validity and the purchase time, and the Deadline variable represents the difference between the end time validity and the purchase time

The *absence3(A)* template states that the activity A has to occur at most twice in the trace. The *absence3(A, Delay,Deadline)* template could state that the occurrence of the activity A has to occur, at most twice, within the time interval delimited by the variables Delay and Deadline with respect to a defined start time.

Let us consider a return train ticket. The ticket can be scanned twice, once for the outbound journey and once for the inbound one. The validity of the ticket defines the time interval, considering the purchase time as starting point. The scan activity should be executed at most twice, and zero times in case where the ticket is refunded.

absence3("scan",Delay,Deadline)

The Delay variable represents the difference between the start time validity and the purchase time, and the Deadline variable represents the difference between the end time validity and the purchase time

The *exclusive_choice(A,B)* constraint template states that either one of the two activities A or B has to eventually be executed. The *exclusive_choice(A,B,Delay, Deadline)* template could state that either the activity A or activity B occurs within time interval delimited by Delay and Deadline.
Let us consider a customer that has purchased a train ticket. The customer can either scan the ticket or refund it, either can be performed within the time validity of the ticket.

exclusive_choice("scan","refund", Delay,Deadline)

The Delay variable represents the difference between the start time validity and the purchase time, and the Deadline variable represents the difference between the end time validity and the purchase time.

5.1.3 Optimisation of the algorithm

In the current implementations of the discovery algorithms, the two temporal variables are calculated every time a constraint is verified. This computational step could be avoided when the Delay Deadline variables are inherited by templates belonging to the same groups.

The branch of the *response(A,B,Delay,Deadline)* subsumptions map in Figure 4.13 shows *chain_response(A,B,Delay,Deadline)* inheriting Delay and Deadline from the *response(A,B,Delay,Deadline)*. This is possible because the three templates belong to the relation template group and how the three templates are calculated, described in paragraph 4.1.3. This implies that the temporal meaning of Delay and Deadline is the same; moreover, if the specialising constraint is verified over the trace, then Delay and Deadline variables would not change in value.

Let us consider the trace [(S,1), (A,1), (B,4), (C,5), (A,7), (B,12), (A,14), (B,19)], when verifying the *response(A,B,Delay,Deadline)* template, Delay and Deadline assume the values of two and five respectively. After the possible specialisation phase, the *chain_response(A,B,Delay, Deadline)* constraint is eventually verified over the trace. The values of Delay and Deadline remains two and five respectively. This example shows that Delay and Deadline can be inherited by templates belonging to the same groups as the values of the variables do not change.

5.1.4 Data-aware constraints

The events considered for this work are in the form of

e(ActivityName, Timestamp)

The information contained for each event are the ActivityName, which is used to define the control-flow of a business process and the Timestamp, which is used to determine the temporal time interval.

It is possible to enrich the model with additional data related to an activity, using a similar approach to the one used for the quantitative temporal variables. The events could be expressed in the form:

e(ActivityName, Data, Timestamp)

Let us consider a model representing the maritime traffic in the gulf of Naples. The position of a ship is determined by using Automatic Identification system messages. It is possible to use the logic proposed in this work to analyse how much time passes between the arrival of a ship at the docks and when it departs. However, it would not be possible to know the type of maritime ship, or it is not possible to determine how long a cargo ship spends docked in the port of Naples.

5.2 Final thoughts

The approach explored in this work to enrich process models with quantitative time constraints has proven viable and reliable. The same approach as discussed in paragraph 5.1.4 can be extended to support data aware constraints in process models. This will allow for more complex analysis of business processes, shifting more towards an automated understanding of them. Discovering data-aware constraints is computationally more costly, therefore addressing the performances of the algorithms will be necessary.

6 References

[1] IEEE Task Force on Process Mining, Process Mining Manifesto. https://www.win.tue.nl/ieeetfpm/downloads/Process%20Mining%20Manifesto.pdf

[2] Pesic, M. (2008). Constraint-based workflow management systems: shifting control to users. Technische Universiteit Eindhoven.

[3] Elena Palmieri, *Learning Declarative Process Models from Positive and Negative Traces*, Alma Mater Studiorum – Università di Bologna (Master thesis) 2020.

[4] Mooney R.J., 1995, *Encouraging experimental results on learning CNF*. https://doi.org/10.1007/BF00994661

[5] O.M.G., Business Process Model and Notation (BPMN), version 2.0, 2011. https://www.omg.org/spec/BPMN/2.0/PDF

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

[7] E. A. Emerson, *Temporal and Modal Logic*, University of Texas at Austin, 1995.

[8] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marella, Sebastian Sardina, *Computing Trace Alignment against Declarative Process Models through Planning*, 2016.

[9] Robert Kowalski, Predicate Logic as a Programming Language, Memo 70, Department of Artificial Intelligence, Edinburgh University, 1973. Also in Proceedings IFIP Congress, Stockholm, North Holland Publishing Co., 1974, pp. 569–574. http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf

[10] Ivan Bratko, Prolog Programming for Artificial Intelligence, 4th Edition. Addison-Wesley 2012, ISBN 978-0-3214-1746-6, pp. I-XXI, 1-673.

[11] L. Console, E. Lamma, P. Mello, M. Milano, *Programmazione Logica e Prolog*, Seconda Edizione UTET, 1997.

[12] A. Colmerauer, P. Roussel, *The birth of Prolog*, 1992.

[13] SWI-Prolog. https://www.swi-prolog.org

[14] CLP(FD) library. https://www.swi-prolog.org/man/clpfd.html

[15] S. Sebahi, Business process compliance monitoring: a view-based approach (Ph.D. thesis), LIRIS, 2012.

[16] D.A. Basin, M. Harvan, F. Klaedtke, E. Zalinescu, Monpoly: monitoring usage-control policies, in: International Conference on Runtime Verification, 2012, pp. 360–364.

[17] D.A. Basin, F. Klaedtke, S. Müller, B. Pfitzmann, *Runtime monitoring of metric first-order temporal properties*, in: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008), LIPIcs, vol. 2, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008, pp. 49–60.

[18] C. Giblin, S. Müller, B. Pfitzmann, From regulatory policies to event monitoring rules: towards model-driven compliance automation, Technical Report, RZ 3662, 2006.

[19] M. Montali, F.M. Maggi, F. Chesani, P. Mello and W. M. P. van der Aalst. *Monitoring Business Constraints with the Event Calculus*, ACM Transactions on Intelligent Systems and Technology, 2013.

[20] M. Montali, F. Chesani, P. Mello, F.M. Maggi, *Towards Data-Aware Constraints in Declare*.

[21] IEEE Task Force on Process Mining: XES Standard Definition (2013).