ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering - DISI

Two year Master Degree in Computer Engineering

LEARNING DECLARATIVE PROCESS MODELS FROM POSITIVE AND NEGATIVE TRACES

Supervisor:
Prof. Federico Chesani

Correlators:
Prof. Paola Mello
Prof. Daniela Loreti

Author
Elena Palmieri

Session III

Academic year 2019-2020

Abstract

In the recent years, the growing number of recorded events made the interest in process mining techniques expand. These techniques make it possible to learn the model of a process, to compare a recent event log with an existing model or to enhance the process model using the information extracted from the log. Most of the existing process mining algorithms only make use of positive examples of a business process in order to extract its model, however, negative ones can bring major benefits. In this work, a discovery algorithm, inspired by the one presented by Mooney in 1995, that takes advantage of both positive and negative sequences of actions is presented in two different versions that return a declarative model connected respectively in disjunctive and conjunctive logic formulas.

Contents

1 Preliminaries **3**

1.1 Process Mining 3

1.1.1 Basic concepts 4

1.1.2 Guiding principles 6

1.2 Declarative process modeling 9

1.2.1 Declare 10

1.2.2 Declare Templates 12

1.3 Prolog 20

1.3.1 Basic concepts 20

1.3.2 SLD Derivation 23

2 The algorithm **25**

2.1 Underlying structure 26

2.1.1 Enhancements 29

2.2 Hierarchy of templates 31

2.3 Choose_Constraint function 41

2.4 Example 46

2.5 DNF vs CNF 48

2.5.1 Backbone algorithm 49

2.5.2 CNF hierarchy of templates 52

2.5.3 CNF choose_constraint function 53

2.5.4 CNF_gain function 54
2.5.5 Example 55
2.6 Optimization 56
2.6.1 Smaller event log 56
2.6.2 Analyzing only a subset of constraints 57
2.6.3 Assert and Retract predicates 59

3 Experimental results **63**
3.1 Controlled event log 63
3.1.1 First set of negative traces 66
3.1.2 Second set of negative traces 68
3.1.3 Third set of negative traces 69
3.1.4 Considerations on the performance 70
3.2 Pap Test screening event log 72
3.3 Real-life event log with no negative traces 75
3.4 Final considerations 78

4 Conclusions and future work **81**
4.1 Future work 83
4.1.1 Optimization of the returned model 83
4.1.2 Properties of the activities 84
4.1.3 Unbounded variables in the model 86
4.1.4 Using CLP 86
4.1.5 Asserting traces 88
4.1.6 Selecting preferred templates 88

Introduction

Process mining techniques [1] allow the analysis of business processes based on event logs. It is in fact possible to extract knowledge from the logs through specific algorithms that will then allow to infer modes, and other information. Process mining techniques are mostly used when a process model is not available, but they have other interesting applications such as the analysis of an event log with the aim of comparing it with an existent model to check whether the latter is compliant with the reality recorded in the log, or the enhancement of an existing process through the information extracted from the log.

The majority of the existing process mining algorithms only use event logs composed of positive traces, sequences of events that represent an execution the process from the beginning to the end; in this work we present an algorithm that also analyzes sequences of actions that represent examples of undesired behavior, called negative traces. Even though it is harder to find event logs that contain a well-structured description of negative examples, they can bring great benefits to the process mining algorithm. For example, they can be used to understand why deviations from the common process model occur or make it possible to specify parts of the process model in a more effective way. The aim of this project is to create a process mining

algorithm that will be able to extract declarative process models from given event logs that represent a particular business process. The models will be expressed in Declare constraints that will either be connected in disjunctive normal form (DNF) or in conjunctive normal form (CNF) formulas. This algorithm will be created drawing inspiration from the one presented by Mooney in his paper *Encouraging experimental results on learning CNF* from 1995 [14] and will have two different versions that will return the two distinct process models.

This work is structured in four chapters. In the first one an overview of the utilized technologies is presented, starting form a more detailed description of process mining and all its possible applications. The other two technologies described in this chapter are the process modelling language that was used to define the constraints that will represent the process model, and the programming language wherewith the program was written. In the second chapter, the proposed algorithm is described in detail, with a particular focus on the differences between the DNF and the CNF version, and on the semantics of the different constraints and their correlations. The third chapter presents a discussion on the experimental results obtained testing the two implementations of the algorithm on the same event log. In particular, there is a focus on the process models obtained and on the temporal performances. The last chapter describes a few possible future additions that could make the algorithm return simpler models in an even shorter amount of time or that could make it possible to infer other information from the event log, other than just the process model, when the information in it contains more detailed data, for example if, other than the name of the activity, it also specifies some properties of the said activity.

Chapter 1

Preliminaries

In this chapter, the three main technologies used in the project will be briefly described, as to give an understanding of its basis.

1.1 Process Mining

Process mining [1] is a research discipline that relates to data science and process management. Its goal is to discover, monitor and improve real processes by extracting knowledge from event logs, and it includes three main family of techniques: process discovery, conformance checking and enhancement.

The growing interest in process mining in the last few years is due to the recording of a growing number of events, that provide detailed information about the history of the processes, and to the business processes' need of improvement and support in competitive and rapidly changing environments.

It is important to note that even though it is its most well-known use, process mining is not limited to control-flow discovery. As previously stated, discovery is one of the three main forms of process mining, but its scope is not limited to control-flow as there are other perspectives that also have significant relevance. Process mining is also not a specific type of data mining, as it requires different representations and algorithms. Finally, process mining is not limited to offline analysis, as a model discovered using existing data can be later applied to running cases.

1.1.1 Basic concepts

The event log is the basis of process mining. Every process mining technique assumes that events are sequentially recorded and that each of them refers to an activity (i.e., a well-defined step in a process) and is related to a specific case (i.e., a process instance). As mentioned above, the first technique of process mining is *discovery*; it consists in taking an event log and creating a model without using a priori information, and it is the most known and most used among the process mining techniques. The second one is *conformance*, a technique that compares an event log with an existent model of that same process. Conformance is quite useful to check whether the reality, as recorded in the log, complies with the model and vice versa. The third and last technique is *enhancement*. It aims at improving or extending the existing process model using the information extracted from the event log.

Process mining covers different perspectives, *control-flow* is the most well-known and it focuses on the order of activities and finds a characterization of all the possible paths that can be expressed, for example, in terms of a Petri net. Another one is the *organizational* perspective, that

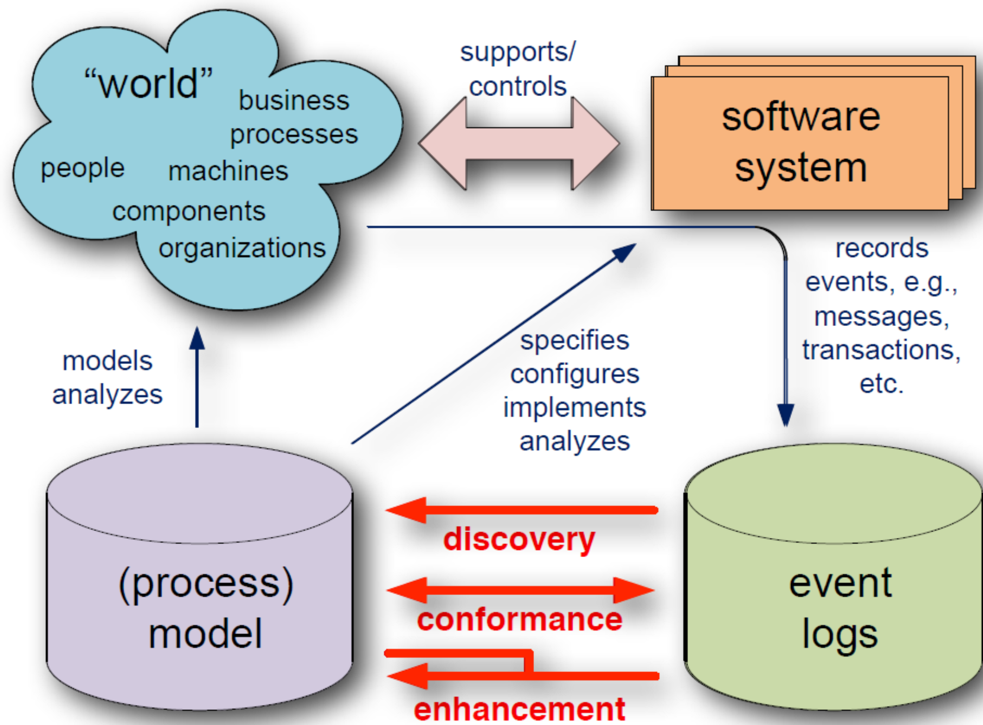

Figure 1.1: Positioning of the three main types of process mining [1]: (a) *discovery*, (b) *conformance* checking, and (c) *enhancement*.

targets information about resources hidden in the log, such as the actors that are involved. Its goal is to structure that information or to show the network that connects it. The *case* perspective focuses on properties of cases, that can be described by the values of the corresponding elements, their path in the process or by the actors involved in it. Lastly, the *time* perspective deals with timing and frequency of events. Indeed, thanks to timestamps the process can be sped up by discovering bottle necks and improved by monitoring the utilization of resources or measuring service levels.

A process mining project can be summarized in 5 stages, as described

in figure 1.2. The first one (Stage 0) concerns justification and planning. In Stage 1 all the event data, models, questions, and objectives are extracted from systems, domain experts and management, thus requiring an understanding of the available data and the domain. Then, in Stage 2, the control-flow model is constructed and linked to the event log, potentially already answering some of the questions and therefore triggering adjustments. In Stage 3 the model should be relatively structured and can be extended by adding other perspectives. The models obtained can then be used to provide operational support (Stage 4).

1.1.2 Guiding principles

In [1] are introduced six guiding principles that should be taken into consideration before applying process mining to real life settings:

- **Event data should be trated as first-class citizen**

  One of the most important parts of process mining is event data. As the quality of the produced model heavily depends on the input, event logs should be treated as first-class citizens. This means that it should be safe to assume that the events in the log actually happened and that their attributes are correct. Furthermore, there should be no missing events in the log and all the recorded ones should have well-defined semantics. Finally, the data should be safe from both a privacy and security point of view.

- **Log extraction should be driven by questions**

  The extraction of the data from the database that contains the event

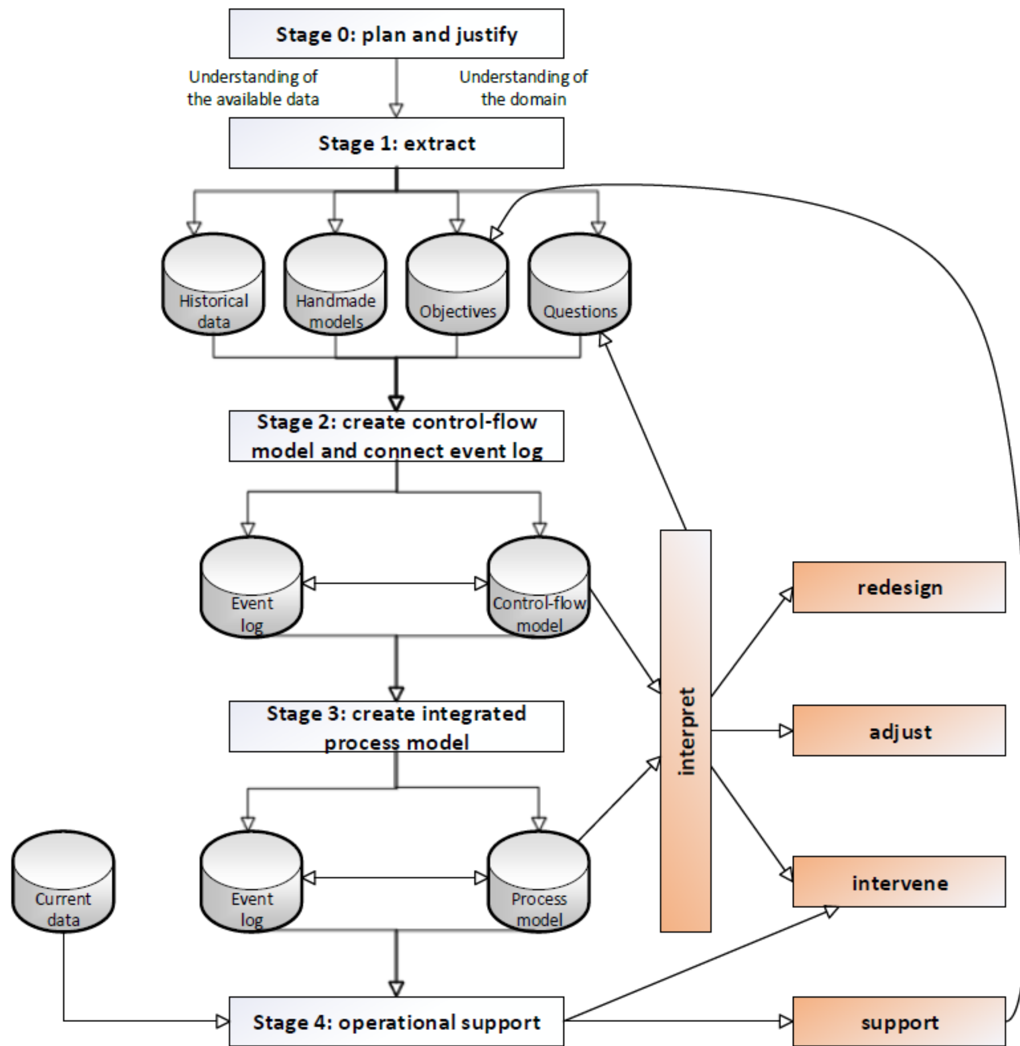

Figure 1.2: The life-cycle model describing a process mining project consisting of five different stages [1]

log should be driven by specific questions. The same data can in fact be used to discover many different process models, each of them centered on a different aspect. For example, the data collected by a delivery company can be used to discover life-cycle of a single delivery or the

one of an order line directed to the same customer. The only way to discover the process model that we are interested in is to extract the data that is meaningful for that particular model.

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

  As control-flow is the most well-known perspective of process mining, basic workflow constructs, such as sequence, parallel routing (AND-splits/joins), choice (XOR-splits/joins) and loops, supported by all mainstream languages, should also be supported by process mining techniques. However, as reported in [1], some of them are not able to deal with concurrency and only support Markov chains, leading to underfitting and/or extremely complex models. It is also desirable to support OR-splits/joins because they provide a compact representation of inclusive decisions and partial synchronizations.

- **Events should be related to model elements**

  Conformance checking and enhancement heavily rely on the relationship between the elements in the model and the events in the log as thanks to it the log can be replayed in the model to find discrepancies. This relationship, though, can be hard to establish and another problem is that events must be related to process instances.

- **Models should be treated as purposeful abstractions of reality**

  The view of reality that a model provides should be treated as a purposeful abstraction of the behavior captured in the event log. The different models derived from an event log can be thought of as different maps that show a view of the same reality in different ways depending,

for example, on the different perspectives or the different levels of granularity and precision. It is therefore important to choose the correct representation and fine-tune it for the intended audience.

- **Process mining should be a continuous process**

  Process mining should be a continuous process that uses both historical event data and current data. Given the dynamic nature of processes, it is in fact not advisable to see process mining just as a one-time activity and instead keep correcting the model based on the new data brings great benefits. Process mining tools can help end users by navigating through processes, projecting dynamic information onto process maps, and providing predictions regarding running cases.

1.2 Declarative process modeling

Imperative process models represent the whole process behavior at once. The most used notation is based on a subclass of Petri Nets called the *Workflow Nets*, but also other implementation exist, such as BPMN. Imperative process models are well-suited for processes in stable business-oriented environments such as production and administrative processes, but they could result in the so-called "spaghetti processes" if used for less static domains as, for example, the healthcare one. Declarative process models [2] describe the processes through the adoption of constraints. The main idea is that every event can happen, except the ones that do not respect the constraints. This makes declarative process models appropriate to represent dynamic domains where the processes can change. The key difference is therefore that imperative process models explicitly specify every

possible sequence of activities, whereas declarative process models only state what is not permitted, so everything that they do not specify is allowed. Figure 1.3 and 1.4 represent a model of the same process respectively through an imperative and a declarative process model.

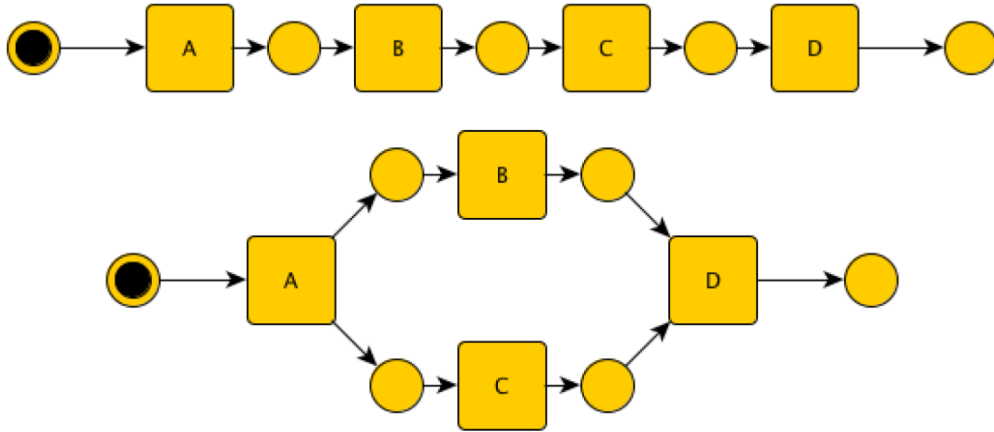

Figure 1.3: Example of an imperative process mining model [3]

1.2.1 Declare

Declare is one of the most well-established declarative process modelling languages. As stated in [4], it provides a set of predefined constraints (repertoire) divided in two main types: existence constraints, that concern the execution of a single activity, and relation constraints, that, on the other hand, involve the execution of two activities. An example of an existence constraint is *Existence(a)*, that specifies that activity *a* has to be executed in every process instance and the constrained task is, therefore, a. *Response(a,b)* is, instead, an instance of relation constraint. It states that if activity a is executed, then activity b must eventually be executed

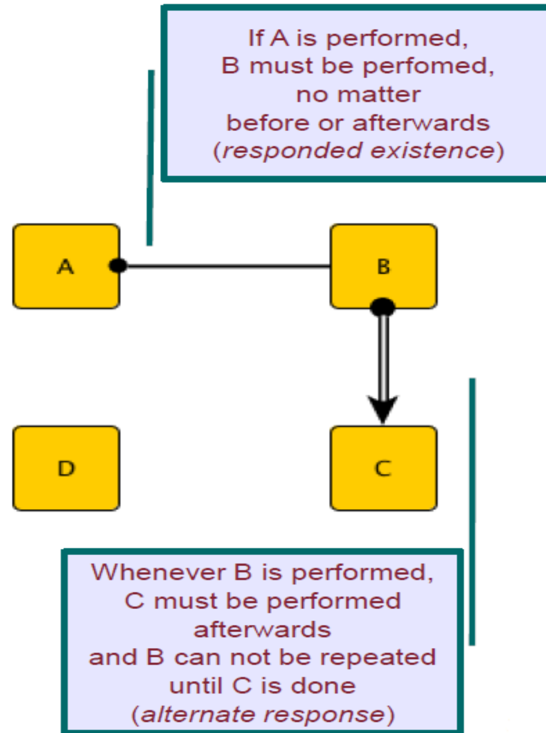

Figure 1.4: Example of a declarative process mining model [3]

after it. For relation constraints, activations and targets are defined; the activation corresponds to the task whose activation triggers the activation of obligations on the completion of the other one, that is for this reason called target. So, still taking as an example the constraint *Response(a,b)*, the activation is a and the target is b.

Formally [4], a template is a predicate $\mathrm{C}_{/\mathrm{n}} \in \mathcal{C}$, where $\mathcal{C}$ is the Declare repertoire and n denotes its number of parameters (arity). A constraint is the application of a template over tasks and is obtained assigning its parameters to elements in $\mathcal{A}$, where $\mathcal{A}$ is called log alphabet and is the set of symbols that identify all the possible tasks and event classes. A parameter assignment γ_{n} is a function $[1,\mathrm{n}] \rightarrow \mathcal{A}$ where $[1,\mathrm{n}]$ is the set of integers ranging from 1 to

n. $\gamma_n(i)$ assigns the i-th parameter of $C_{/n}$ to a task in A, in compliance with the positional notation of the parameters of predicates. For the constraint *Response(a,b)* then, $\gamma_2(1) = a$ and $\gamma_2(2) = b$. From this, we can now define a declarative process model as a tuple $M = (A, C, \Gamma)$ where: A is a finite non-empty set of tasks, C is a finite non-empty repertoire of templates and $\Gamma = \{C\gamma_{/n}(i) : C_{/n} \in C, \gamma_n(i) : [1, \ldots, n] \to A, i \geq 0\}$ is a set of well-defined constraints that corresponds to the set of all the constraints that derive from the instantiation of every template $C_{/n} \in C$ with every possible assignment to a task in A.

1.2.2 Declare Templates

Below is given the definition of all the Declare templates that are used in this thesis. Other than existence and relation templates, choice templates, which description can be found in [5], and negation templates are introduced; the latter define a negative relation between activities. This was necessary because in Declare the NOT operator is not defined. For every template it will also be specified its Linear Temporal Logic (LTL) expression.

Linear Temporal Logic

LTL is a special kind of logic and is used for describing sequences of transitions between states in reactive systems [6]. A well-formed formula p over E, where E is a subset of all the possible events, is a function p: E^* $\to$ {true, false}, where E is the set of all the possible events. Let $\sigma \in E^*$ be a trace, if p is a well-formed formula and it holds that $p(\sigma)$ = true than we say that p satisfies σ, denoted by $\sigma \models p$. If p and q are well-formed

formulas, then also true, false, $!p$, $p \wedge q$, $p \vee q$, $\Box p$, $\Diamond p$, $\bigcirc p$, $p\mathrm{U}q$ and $p\mathrm{W}q$ are well-formed formulas over E.

The LTL semantics are defined as:

- **position:** $\sigma \models e$ if and only if $e = \sigma[1]$, for $e \in$ E. Note that σ[i] denotes the i-th element of the trace, i.e. if $\sigma = (\mathrm{e_1}, \mathrm{e_2}, ..., \mathrm{e_n})$, then $\sigma[\mathrm{i}] = \mathrm{e_i}$.
- **not (!):** $\sigma \models !p$ if and only if not $\sigma \models p$.
- **and ($\wedge$):** $\sigma \models p \wedge q$ if and only if $\sigma \models p$ and $\sigma \models q$.
- **or ($\vee$):** $\sigma \models p \vee q$ if and only if $\sigma \models p$ or $\sigma \models q$.
- **next ($\bigcirc$):** $\sigma \models \bigcirc p$ if and only if $\sigma^{2\rightarrowtail} \models p$. $\sigma^{\mathrm{i}\rightarrowtail}$ indicates the suffix of σ starting at σ[i].
- **until (U):** $\sigma \models p\mathrm{U}q$ if and only if $(\exists_{1 \leq \mathrm{i} \leq \mathrm{n}} : (\sigma^{\mathrm{i}\rightarrowtail} \models q \wedge (\forall_{1 \leq \mathrm{j} \leq \mathrm{i}} : \sigma^{\mathrm{i}\rightarrowtail} \models p)))$.

It is also possible to use the following abbreviations:

- **implication ($p \Rightarrow q$):** for $!p \vee q$.
- **equivalence ($p \iff q$):** for $(p \wedge q) \vee (!p \wedge !q)$.
- **true (true):** for $p \vee !p$.
- **false (false):** for !true.
- **eventually ($\Diamond$):** for $\Diamond p = \mathrm{true}\mathrm{U}p$.

- **always ($\Box$):** for $\Box p = !\Diamond !p$.
- **weak until (W):** for $p\mathrm{W}q = (p\mathrm{U}q) \vee (\Box p)$.

Explained in other words, the operator always ($\Box p$) specifies that p holds at every position in the trace, eventually ($\Diamond p$) that p holds at least once in the trace, next ($\bigcirc p$) that p holds in the next element of the trace, until ($p\mathrm{U}q$) that there is a position in the trace in which q holds and p holds in all the previous ones, and weak until ($p\mathrm{W}q$) that is similar to until but does not require q to ever become true.

Existence templates

These templates only involve one activity and they define the cardinality or the position of that activity in the trace.

The template *existence(A)* indicates that activity A must be executed at least once in the trace. A lower bound for the number of occurrences of A can be specified creating the templates *existence2(A)* and *existence3(A)* that respectively state that the activity A has to be executed at least twice and three times. The *absence(A)* template has the opposite semantics with respect to *existence*, activity A cannot be present in the trace. In this case, we can include an upper bound that will state the maximum number of occurrences of the activity. In *absence2(A)* and *absence3(A)* the upper bounds are respectively 2 and 3, meaning that in the former A can be executed 0 or 1 times and in the latter from 0 up to 2 times. Moreover, the templates *exactly1(A)* and *exactly2(A)* denote that activity A has to be executed exactly the specified number of times. Finally, the template *init(A)* asserts that the trace must start with activity A and its dual, *last(A)*, that

template	LTL expression
existence(A)	$\Diamond(\mathrm{A}, \mathrm{t_c})$
existence2(A)	$\Diamond((\mathrm{A}, \mathrm{t_c}) \wedge \bigcirc(\mathit{existence(A)}))$
existence3(A)	$\Diamond((\mathrm{A}, \mathrm{t_c}) \wedge \bigcirc(\mathit{existence2(A)}))$
absence(A)	$!\mathit{existence(A)}$
absence2(A)	$!\mathit{existence2(A)}$
absence3(A)	$!\mathit{existence3(A)}$
exactly1(A)	$\mathit{existence(A)} \wedge \mathit{absence2(A)}$
exactly2(A)	$\mathit{existence2(A)} \wedge \mathit{absence3(A)}$
init(A)	$((\mathrm{A}, \mathrm{t_s}) \vee (\mathrm{A}, \mathrm{t_x}))\mathrm{W}(\mathrm{A}, \mathrm{t_c})$
last(A)	$\Diamond(\mathrm{A} \wedge \bigcirc!\mathrm{true})$

Table 1.1: LTL expression of existence templates ($\mathrm{t_c}$, $\mathrm{t_s}$ and $\mathrm{t_x}$ are event types such that $\mathrm{t_c}$ = *completed*, $\mathrm{t_s}$ = *started* and $\mathrm{t_x}$ = *cancelled*) [6]

template	LTL expression
choice(A,B)	$\Diamond\mathrm{A} \vee \Diamond\mathrm{B}$
exclusive_choice(A,B)	$(\Diamond\mathrm{A} \vee \Diamond\mathrm{B}) \wedge !(\Diamond\mathrm{A} \wedge \Diamond\mathrm{B})$

Table 1.2: LTL expression of choice templates [6]

the trace must end with it.

Choice templates

The choice templates are only two, they describe a relation between two activities and have, therefore, activities A and B as parameters.

The *choice(A,B)* template indicates that activities A and B eventually occur in each process instance. There is no restriction, so it is possible that both occur or that only one of them does. The *exclusive_choice(A,B)* template is more limiting because it forbids A and B to both occur in the same process instance.

Relation templates

A relation template defines a dependency between two activities, they all have activities A and B as parameters.

The *responded_existence(A,B)* template specifies that if activity A is executed, then activity B needs to be executed as well, either before or after activity A. It is important to note that a constraint that derives from this template will be satisfied if the activity A is never executed. The definition of *co-existence(A,B)* is quite similar to the previous one, but it holds for both activities: if activity A is executed, then activity B has to be executed as well and, vice versa, if B is executed, then also A needs to be executed.

The previous templates did not consider the order of the activities, there are though templates that specify it. *response(A,B)* imposes that every time the activity A is executed, activity B needs to be executed after it. There can still be other executions of A and other activities in between the two that are taken into consideration. *precedence(A,B)*, on the other hand, states that if activity A is executed, then it must be preceded by activity B. Again, there can be multiple executions of any activity between A and B. The combination of these last two templates defines a bi-directional execution order of two activities and is called *succession(A,B)*. In order for it to hold, both *response(A,B)* and *precedence(A,B)* must be satisfied.

Templates *alternate_response(A,B)*, *alternate_precedence(A,B)* and *alternate_succession(A,B)*, are a stronger version of the templates that were just described. In *alternate_response(A,B)* after the execution of activity A an activity B needs to be executed, but there cannot be other executions of activity A before the one of activity B. Similarly,

template	LTL expression
responded_existence(A,B)	$\Diamond(\mathrm{A}, \mathrm{t_c}) \Rightarrow \Diamond(\mathrm{B}, \mathrm{t_c})$
co-existence(A,B)	$\Diamond(\mathrm{A}, \mathrm{t_c}) \iff \Diamond(\mathrm{B}, \mathrm{t_c})$
response(A,B)	$\Box((\mathrm{A}, \mathrm{t_c}) \Rightarrow \Diamond(\mathrm{B}, \mathrm{t_c}))$
precedence(A,B)	$(!((\mathrm{B}, \mathrm{t_s}) \vee (\mathrm{B}, \mathrm{t_c}) \vee (\mathrm{B}, \mathrm{t_x})))\mathrm{W}(\mathrm{A}, \mathrm{t_c})$
succession(A,B)	*response(A,B)* $\wedge$ *precedence(A,B)*
alternate_response(A,B)	*response(A,B)* $\wedge \Box((\mathrm{A}, \mathrm{t_c}) \Rightarrow \bigcirc($*precedence(B,A)*$))$
alternate_precedence(A,B)	*precedence(A,B)* $\wedge \Box((\mathrm{B}, \mathrm{t_c}) \Rightarrow \bigcirc($*precedence(A,B)*$))$
alternate_succession(A,B)	*alternate_response(A,B)* $\wedge$ *alternate_precedence(A,B)*
chain_response(A,B)	*response(A,B)* $\wedge \Box((\mathrm{A}, \mathrm{t_c}) \Rightarrow \bigcirc(\mathrm{B}, \mathrm{t_s}))$
chain_precedence(A,B)	*precedence(A,B)* $\wedge \Box(\bigcirc(\mathrm{B}, \mathrm{t_s}) \Rightarrow (\mathrm{A}, \mathrm{t_c}))$
chain_succession(A,B)	*chain_response(A,B)* $\wedge$ *chain_precedence(A,B)*

Table 1.3: LTL expression of relation templates ($\mathrm{t_c}$, $\mathrm{t_s}$ and $\mathrm{t_x}$ are event types such that $\mathrm{t_c}$ = *completed*, $\mathrm{t_s}$ = *started* and $\mathrm{t_x}$ = *cancelled*) [6]

in *alternate_precedence(A,B)*, if activity A is executed, activity B must precede it but between the execution of the two, other activities can be executed, except activity A. *alternate_succession(A,B)* is again defined as the combination of *alternate_response(A,B)* and *alternate_precedence(A,B)*.

Lastly, the templates *chain_response(A,B)*, *chain_precedence(A,B)* and *chain_succession(A,B)* are the strongest version. *chain_response(A,B)* states that if activity A is executed, activity B needs to be executed right after it; *chain_precedence(A,B)* that if activity A is executed, it must be directly preceded by activity B. So, the two activities have to be next to each other. *chain_succession(A,B)* is, as usual, the combination of the other two chain templates.

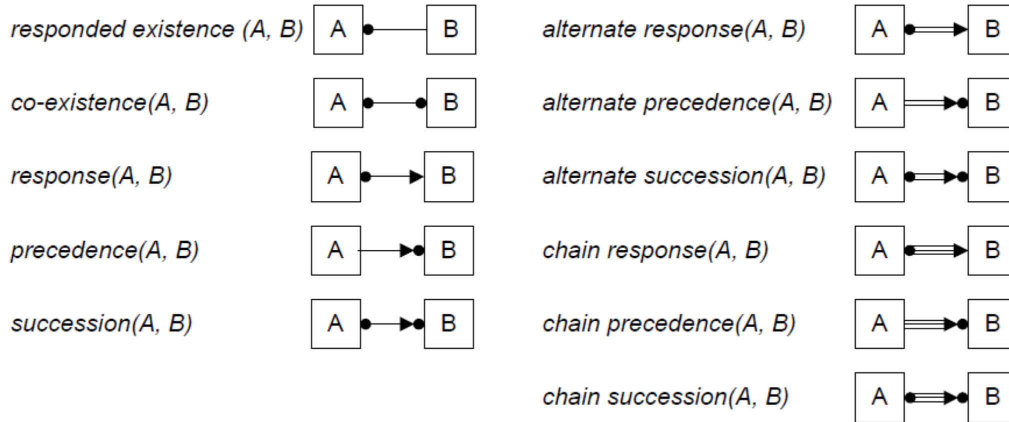

Figure 1.5: Notation for the relation templates [3]

Negation templates

Negation templates are the negated version of the relation templates; please note that these templates do not correspond to the logical negation, with respect to LTL, of the relation templates because the NOT operator is not defined in Declare.

The *not_responded_existence(A,B)* template specifies that if activity A is executed, then activity B must never be executed in the trace. As a consequence of this not being the logical negation of *responded_existence(A,B)*, if A never occurs, then both templates hold, meaning that one does not exclude the other. *not_co-existence(A,B)* applies the previous template from A to B and from B to A.

not_response(A,B) states that if activity A is executed, then activity B cannot be executed anymore. Likewise, *not_precedence(A,B)* imposes that activity A must not be preceded by activity B. These two templates are, as usual, combined to obtain *not_succession(A,B)*.

template	LTL expression
not_responded_existence(A,B)	$\Diamond(\mathrm{A}, \mathrm{t_c}) \Rightarrow !(\Diamond(\mathrm{B}, \mathrm{t_c}))$
not_co-existence(A,B)	*not_responded_existence(A,B)* $\wedge$ *not_responded_existence(B,A)*
not_response(A,B)	$\Box((\mathrm{A}, \mathrm{t_c}) \Rightarrow !(\Diamond((\mathrm{B}, \mathrm{t_s}) \vee (\mathrm{B}, \mathrm{t_c}))))$
not_precedence(A,B)	$\Box(\Diamond(\mathrm{B}, \mathrm{t_s}) \Rightarrow (!(\mathrm{A}, \mathrm{t_c})))$
not_succession(A,B)	*not_response(A,B)* $\wedge$ *not_precedence(A,B)*
not_chain_response(A,B)	$\Box((\mathrm{A}, \mathrm{t_c}) \Rightarrow \bigcirc(!(\mathrm{B}, \mathrm{t_s})))$
not_chain_precedence(A,B)	$\Box(\bigcirc(\mathrm{B}, \mathrm{t_s}) \Rightarrow !(\mathrm{A}, \mathrm{t_c}))$
not_chain_succession(A,B)	*not_chain_response(A,B)* $\wedge$ *not_chain_precedence(A,B)*

Table 1.4: LTL expression of negation templates ($\mathrm{t_c}$ and $\mathrm{t_s}$ are event types such that $\mathrm{t_c}$ = *completed* and $\mathrm{t_s}$ = *started*) [6]

According to the *not_chain_response(A,B)* template, every time activity A is executed, it cannot be directly followed by activity B and in order for *not_chain_precedence(A,B)* to hold, activity A must never be directly preceded by activity B. Combined together these two templates define *not_chain_succession(A,B)*.

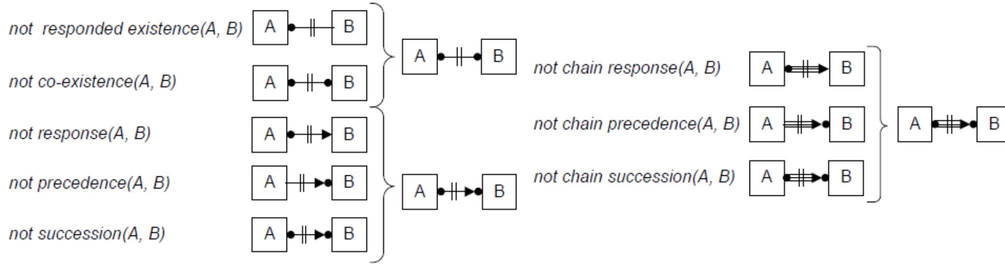

Figure 1.6: Notation for the negation templates [3]

1.3 Prolog

Prolog is a logic programming language designed in 1972 by Alain Colmerauer and Philippe Roussel. Its name derives from PROgramming in LOGic and it is based on Robert Kowalski's interpretation of Horn clauses [7]. Prolog has its roots in first-order logic and is intended as a declarative programming language, meaning that the program logic is expressed in terms of relations, that are represented as facts and rules. For more information about Prolog see [8], [9] and [10].

Prolog is the chosen programming language for the project because, thanks to its declarative nature, it is very well suited to represent the different traces and to find patterns among them. Specifically it was used the SWI-Prolog implementation [11], that offers a rich set of features and libraries for constraint programming.

1.3.1 Basic concepts

Formally, the Prolog language is a special case of logic programming. A Prolog program is defined by a set of clauses defined as following:

(clause1) A.

(clause2) A :- B_1, B_2, ..., B_n.

(clause3) :- B_1, B_2, ..., B_n.

Clause1 is a *fact*, clause2 is a *rule* and clause3 is a *goal*. A and B_i are atomic formulas, A is the *head* of the clause and B1, B_2, ..., B_n is the *body*

of the clause. The symbol "," indicates the conjunction and ":-" the logic implication in which A is the consequent and B_1, B_2, ..., B_n the antecedent. An *atomic formula* is a formula such as:

$p(t_1, t_2, ..., t_n)$

where p is a predicative symbol and $t_1, t_2, \ldots, t_n$ are terms. A term can be a constant (as they were in the previous example: bob, joe, tyler), a variable (identified with a capital letter or preceded by the symbol "_": X, Variable1, _x) or another function that can have its own arguments, which are, again, terms (f(a), f(g(X)), f(a, b)). A computation corresponds to the attempt to prove that a formula logically follows a program, i.e., that a formula is a theorem. The computation starts running a query (goal) and it aims at finding a substitution for its variables so that given a program P and the query:

$:- p(t_1, t_2, ..., t_m).$

If $X_1, X_2, \ldots, X_n$ are the variables in $t_1, t_2, \ldots, t_m$, the meaning of the query is

$\exists X1, \exists X2, ..., \exists Xn\ p(t_1, t_2, ..., t_m)$

And the substitution is

$\sigma = \{X_1/s_1, X_2/s_2, ..., X_n/s_n\}$

where s_i are terms, and so that $P \models [p(t_1, t_2, \ldots, t_m)]\sigma$

It is important to verify that in a unification of a variable X and a composed term S, the latter does not contain the variable X. This would invalidate both the termination of the algorithm and the correctness of the solution. For example, the unification between p(X,X) and p(Y, f(Y)) would have the substitution [X/Y, X/f(Y)] meaning that Y/f(Y), which would make the computation enter in a loop without termination.

A Prolog program is a set of Horn clauses, a disjunction of literals in which at most one of them is positive, that represent facts, rules and goals. The facts express the objects and the relations among them; below are a few examples of facts:

```
father(bob, joe).
father(bob, mary).
father(joe, tyler).
```

The rules are expressed on objects and relations and are divided in head and body, following the if – then construct indicated by the symbol of implication ":-".

```
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

The head is the part on the left of the implication symbol (grandfather(X,Y) in the example above) and the body is the part on the right ("father(X,Z), father(Z,Y).", where the symbol "," indicates a conjunction). Goals are clauses that do not have a head and are based on the knowledge defined in the facts and in the rules.

```
:- grandfather(X,Y).
```

The goal is proved unifying it with the heads of the clauses in the program, if a substitution for which the comparison succeeds exists then there are two alternatives: if the found clause is a fact the program ends, if it is a rule, then its body has to be verified as well. In the example, the goal unifies with the rule

```
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

So now we have to also verify the body of this rule. Starting from father(X,Y), it can be seen that it unifies with all the three facts, but Prolog always starts the unification with the fact that is written above the others in the program's code, so in this case X is unified with bob and Z with joe. Now we are left with the second part, but as Z is already unified with joe, only the fact father(joe,tyler) is a good match and the variable Y is therefore unified with tyler. Both formulas are now satisfied, so the original goal is satisfied as well, and the given answer is:

```
grandfather(bob, tyler).
```

1.3.2 SLD Derivation

A SLD derivation for a goal G_0 from the set of defined clauses P is a sequence of goal clauses $G_0, \ldots, G_n$, a sequence of variants of clauses of the program $C_1, \ldots, C_n$, and a sequence of most general unifier substitutions (we always want the most general substitution, a substitution σ_1 is more general than

another substitution σ_2 if there exists a third one σ_3 so that $\sigma_2 = \sigma_1\sigma_3$) σ_1, $\ldots$, σ_n so that G_{i+1} is derived from G_i and C_{i+1} through the substitution σ_n. There exist three different types of derivation:

- **Success** if for n finite G_n is equal to the empty clause.
- **Finite failure** if for n finite it is not possible to derive a new solver from G_n and G_n is not equal to the empty clause.
- **Infinite failure** if it is always possible to derive new solvers, all different from the empty clause.

There are, therefore, two forms of non-determinism; the first due to the selection of an atom from the goal to unify with the head of the clause, that is solved defining a calculation rule. This rule does not affect correctness nor completeness, only efficiency; given a program, its success set does not depend on the calculation rule used in the SLD resolution. The second form depends on the choice of the clause that the program uses in a resolution step, and is solved defining a research strategy. If more solutions exist for the same goal, the SLD resolution must be able to find them all and guarantee completeness.

Chapter 2

The algorithm

One of the main differences between this process mining algorithm and the existing ones is precisely the use of negative traces. In this thesis with the term *negative traces* we refer to all the traces that do not follow the common path; the event log is considered to be split into positive traces and these other traces like in Machine Learning. The other traces will, for convenience, be referred to as negative traces, or negative examples. As very few other declarative process discovery approaches, such as [12], we assume that the log contains both positive and negative examples; meaning that, besides the allowed cases, also instances of undesired behaviors can be exploited. This allows to better control the degree of generalization of the resulting model, as well as improve its simplicity by ignoring the parts of the model that are not significant to discriminate positive and negative behaviors. The reason why the majority of the process mining algorithms do not use negative traces is that in most of the real-life situations, distinguish positive and negative cases in the input log is a very hard and time-consuming task and in some cases it might not even be possible, so it would be easier to work as if the negative

examples did not exist. This set constitutes a sort of "upside-down world", specular to the real world of positive, common and allowed cases. These can be used to understand the reasons why deviations from the common process model occur. This information is useful not only to clarify what should be deemed compliant with the model and what should not, but also to specify parts of the business process in a more synthetic and effective way, for example a set of constraints that results form positive examples may be substituted by a single one extracted from negative traces. See [13] for a full discussion on this topic.

2.1 Underlying structure

In his paper [14], Mooney proposed two algorithms that learn-first order Horn clauses, one that returns the final result as a disjunctive normal form (DNF) expression and the other that returns it as a conjunctive normal form (CNF) expression; for now, we will focus on the former. This algorithm, presented in pseudocode below, was chosen as the backbone structure of the project. It is important to state that even though Mooney's algorithms were thought for Horn clauses, the basic algorithms were heuristic covering algorithms for learning DNF or CNF; here they were adapted to process mining, as described in the next subsection.

The algorithm is composed of two cycles, the outer one takes in input the list of positive examples and the list of negative examples and ends when the list of positive examples becomes empty. It makes a copy of the two sets and then calls the inner cycle, that ends, instead, when the copied list of negative examples is empty. The aim of this second cycle is to construct a

Algorithm 1 DNF learner by Mooney

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 constraint C that maximizes the DNF-gain.

Add the constraint C to Term.

Remove from Neg all the examples that do not satisfy C.

Remove from Pos2 all the examples that do not satisfy C.

Add Term as one term of DNF

Remove from Pos all the examples that satisfy Term.

Return DNF.

term composed by constraints in AND that excludes all the negative traces while covering at least one of the positives, and that will then be added in OR to the model. Please note that this is not an extension of the Declare language; saying that the terms will be in OR means that the produced model will be presented in DNF, not that the OR is present in the language. The internal representation of the model is simply implemented as a list of lists, where the different lists will be presented in OR between each other and the elements in the list in AND. Inside the second cycle, the algorithm calls a function to choose the most convenient constraint and adds it, in AND, to the term that it is building. Then, it removes from its copy of the lists of positive and negative examples the ones that do not satisfy the new constraint. After finding a term containing constraints that exclude all the negative traces, or, in other words, when the list of negative examples becomes empty, the inner loop ends, the term is added in OR to the previous ones and all the positive traces that satisfy the new term are removed from the original list of positive examples. When this list becomes empty, the algorithm ends and returns the model. As the focus of the inner cycle is to find a term that excludes all the negative traces, probably some of the positive ones will be excluded as well. The outer cycle eliminates the covered traces from the list of positive traces to make it possible for the inner cycle to find a second term that will focus on finding different constraints. The DNF gain, taking into consideration the number of positive traces covered, is higher for the constraints that cover more positive traces, while excluding the same number of negative traces; so changing the list, the chosen constraints will change too.

2.1.1 Enhancements

To enhance the quality of the model given by the algorithm a few modifications were made. Every term returned by the inner cycle is added to it only if it covers at least one positive trace, otherwise it is discarded as it would be redundant. The fact that the algorithm removes the positive traces covered by the term from the list of the positive traces makes this control quite easy to do as it possible to compare the old list to the new one and, if the two are the same, it means that no positive trace has been removed so no positive trace satisfies the new term.

Another major change with respect to Mooney's algorithm is that this one allows some traces not to be covered by the model. It is in fact possible that a few traces, positive or negative, are in contrast with the other ones and including them in the model would be impossible. A simple example, that should not occur in a real event log, would be the case of the same trace included in both the sets of positive and negative examples. In this case it would obviously be impossible to create a model that includes the positive one and simultaneously excludes the negative one. To avoid the program to fail and not return any model, a control was added to make sure that if the function of the inner cycle that chooses the next constraint to add to the term fails, i.e., there are no more constraints that can be added and, therefore, the negative traces left are not possible to exclude, these remaining examples are removed and returned. The same thing happens if there are no more terms that can cover the remaining positive traces; the inner cycle aims to cover as many positive traces as possible, so if the term that it returns covers no positive traces at all, it means that they cannot be covered in any way and that they have to be discarded. As for the negative traces, they are

Algorithm 2 DNF version of the enhanced algorithm

```
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).
```

stored in a variable that keeps track of their previous presence in the list. When the algorithm comes to an end and returns the model, the discarded traces are printed on the screen so the user can know that it would not have been possible to discover a process model given the complete log, and that the given one only describes a part of the it.

2.2 Hierarchy of templates

Before describing the details of the algorithm itself, it is necessary to explain how the different templates among which the choose_constraint function will look for the next constraint to add to the term, are organized. It is possible to identify templates that are more specific than others, i.e., that pose a tighter constraint on the traces. For example, the existence(A) template is one of the most general ones as it only states that a certain activity has to be executed in the trace, without specifying how many times or the time of its execution. On the other hand, the template chain_succession(A,B) is one of the most specialized as it imposes many conditions on the two activities.

In [4] can be found a subsumption map of Declare templates, shown in Figure 2.1, that makes the general structure of the hierarchy easier to understand.

The upper part, Figure 2.1(a), illustrates the subsumption hierarchy of existence templates. Given two n-ary templates, we say that *C is subsumed by C'*, written $C \sqsubseteq C'$, if for every trace t $\in A^*$ and every parameter assignment γ_n from the parameters of C to tasks in A, whatever t complies with the instantiation of C with γ_n, then t also complies with the instantiation of *C'* with γ_n. Following this definition, a template is more

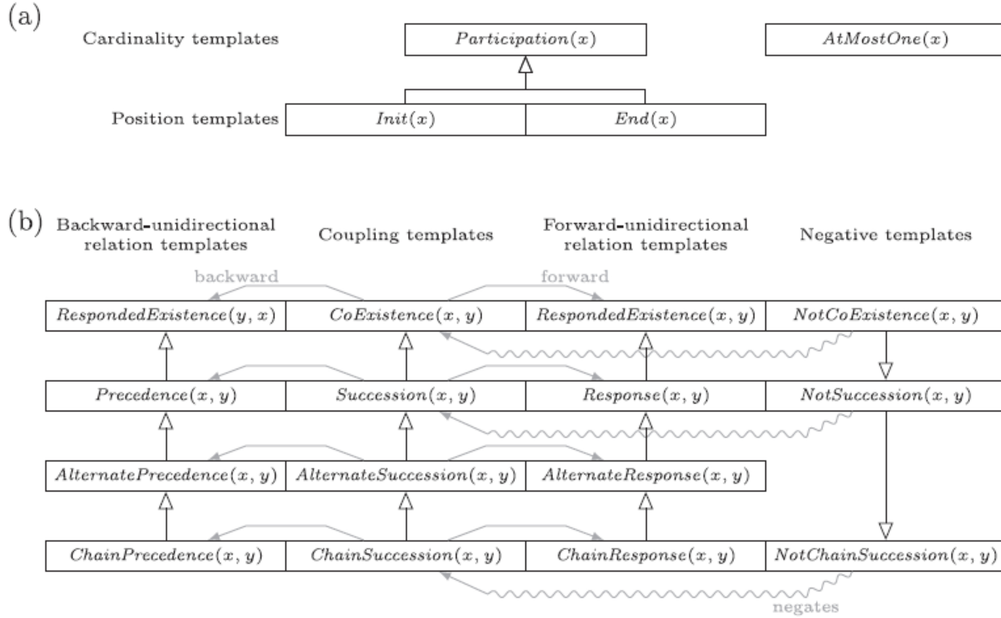

Figure 2.1: Subsumption map of Declare templates [4]. Please note that *Participation(x)* corresponds to the template described as *existence(X)*, *End(x)* to *last(X)* and *AtMostOne(x)* to *absence2(X)*

specialized than another if it is subsumed by it; for example, response(a,b) $\sqsubseteq$ responded_existence(a,b) so response(a,b) is more specialized than responded_existence(a,b). On the other hand, responded_existence(a,b) is more general than response(a,b). The templates are indicated in solid boxes and the subsumption is drawn with a line that starts from the subsumed template and ends in the subsuming one, with an empty triangular arrow.

Figure 2.1(b) shows that responded_existence(x,y) directly subsumes response(x,y) and precedence(y,x). Both constraints strengthen the conditions imposed by responded_existence by specifying that not only the target must occur, but also in which relative position in the trace (after or before the activation). However, as the role of activation and target are

swapped in precedence with respect to responded_existence, the latter and Response belong to the type of *forward unidirectional relation templates*, whereas precedence is a *backward unidirectional relation template.*

The direct child templates of response and precedence are alternate_response and alternate_precedence. The concept of alternation strengthens the parent template by adding the condition that between pairs of activation and targets, no other activation can occur. The same concept applies between alternate_response and alternate_precedence on one side and chain_response and chain_precedence on the other.

The conjunction of a forward unidirectional relation template and a backward unidirectional relation template belonging to the same level of the subsumption hierarchy generates the so-called *coupling templates*: succession(x,y). The coupling template co_existence(x,y) is equal to the conjunction of responded_existence(x,y) and responded_xistence(y,x). For every coupling template C, the functions *fw*(C) and *bw*(C) are defined and return respectively the related forward unidirectional relation template and backward unidirectional relation template. Hence *fw*(succession(x,y)) = response(x,y) and *bw*(succession(x,y)) = precedence(x,y). In Figure 2.1(b) these functions are indicated by gray arcs labeled as forward and backward.

The negative templates negate the corresponding coupling templates and the subsumption hierarchy gets reverted so that not_co_existence is subsumed by not_succession that is subsumed by not_chain_succession.

Remembering that the templates used in this project are more than the ones in Figure 2.1, the hierarchy was created thinking of it as a tree that has more than one starting node. These correspond to the most general templates and are existence(X), responded_existence(X),

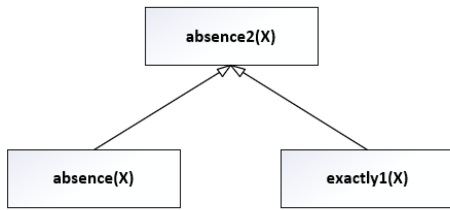

Figure 2.2: Subsumption map of *absence2(X)* template.

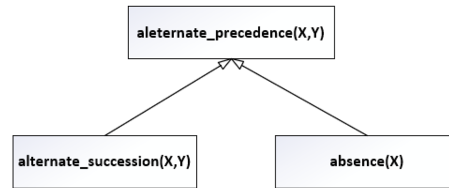

Figure 2.3: Subsumption map of *alternate_precedence(X, Y)* template.

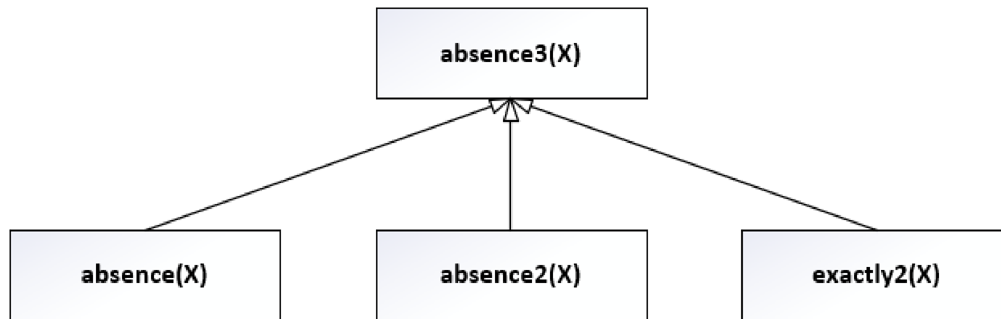

Figure 2.4: Subsumption map of *absence3(X)* template.

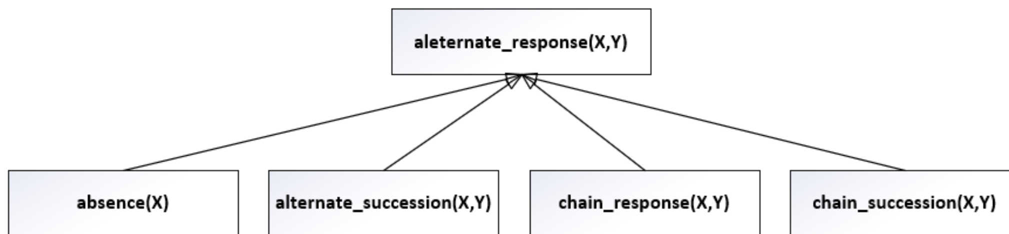

Figure 2.5: Subsumption map of *alternate_response(X, Y)* template.

absence3(X), choice(X,Y) and not_chain_succession(X,Y). The specialization of these templates creates the different branches of the tree.

In Figures 2.2 to 2.37 are shown all the subsumption maps used in

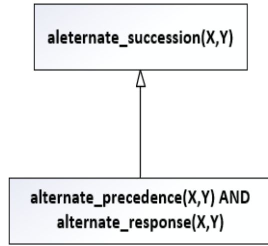

Figure 2.6: Subsumption map of *alternate_succession(X, Y)* template.

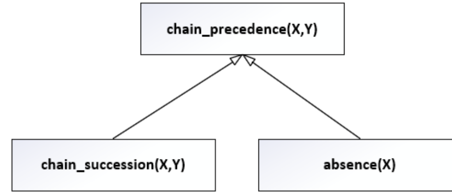

Figure 2.7: Subsumption map of *chain_precedence(X, Y)* template.

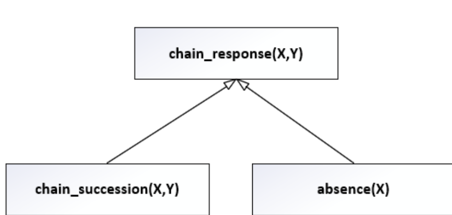

Figure 2.8: Subsumption map of *chain_response(X, Y)* template.

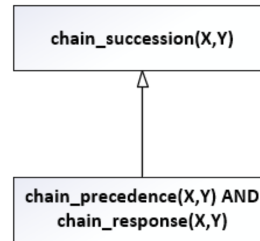

Figure 2.9: Subsumption map of *chain_succession(X, Y)* template.

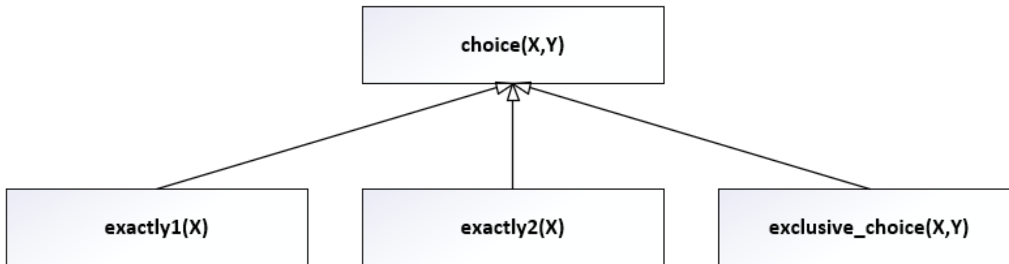

Figure 2.10: Subsumption map of *choice(X, Y)* template.

the project. This hierarchy was taken from [13], it is not as straight forward as the one described above, but it makes it possible to provide simpler models. For example, this hierarchy allows to take the template

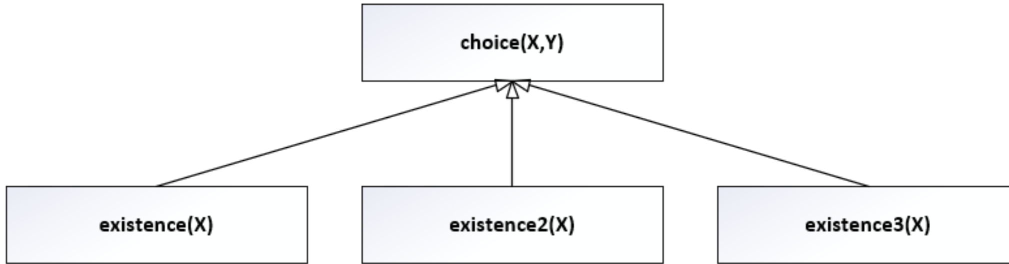

Figure 2.11: Subsumption map of *choice(X, Y)* template.

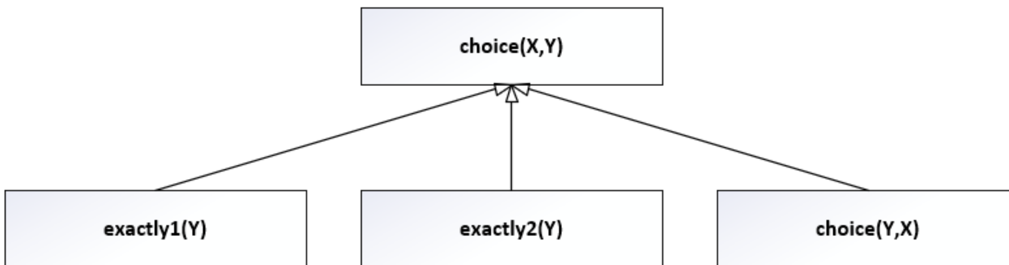

Figure 2.12: Subsumption map of *choice(X, Y)* template.

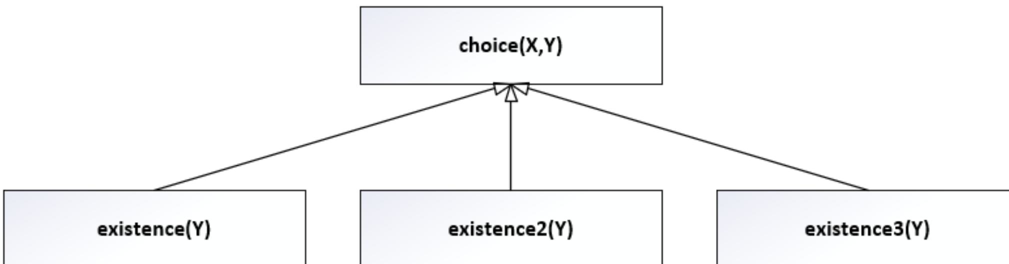

Figure 2.13: Subsumption map of *choice(X, Y)* template.

chain_precedence(X, Y) directly as a specialization of *precedence(X, Y)* without inserting also *alternate_precedence(X, Y)* in the model. A downside of this is that every template will have a higher number of subsumed ones, possibly slowing down the computation.

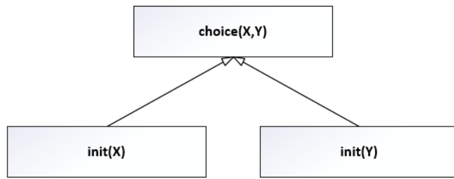

Figure 2.14: Subsumption map of *choice(X,Y)* template.

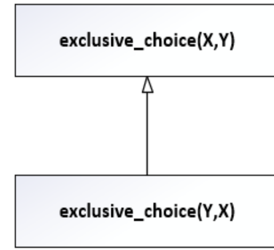

Figure 2.15: Subsumption map of *exclusive_choice(X,Y)* template.

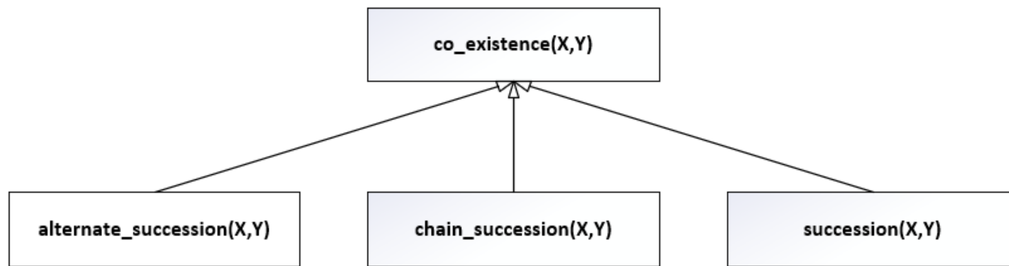

Figure 2.16: Subsumption map of *co_existence(X,Y)* template.

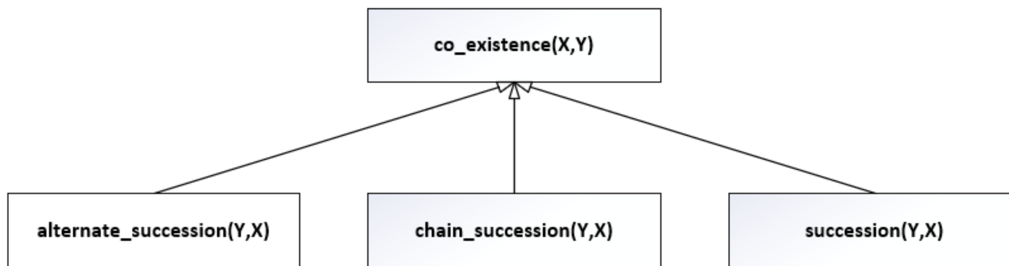

Figure 2.17: Subsumption map of *co_existence(X,Y)* template.

It was not possible to create a single tree as it would have been unreadable because of its dimension, moreover some of the maps had to be split into multiple parts for the same reason.

This allows all the specialized constraints to be found just by using the

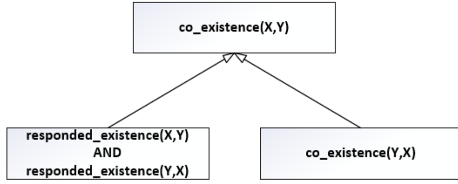

Figure 2.18: Subsumption map of *co_existence(X, Y)* template.

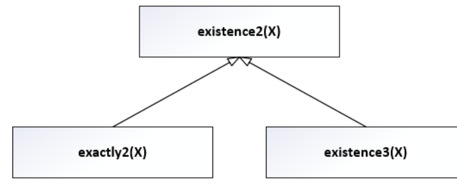

Figure 2.19: Subsumption map of *existence2(X, Y)* template.

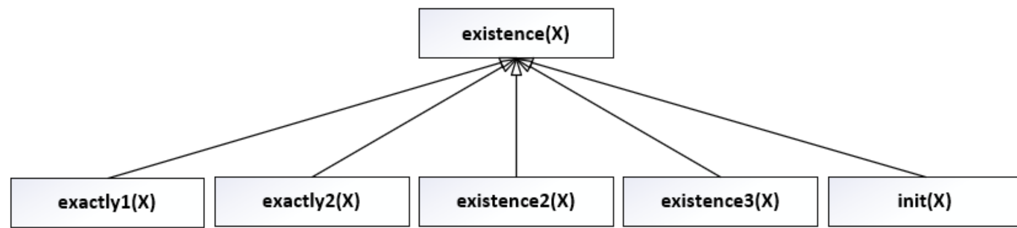

Figure 2.20: Subsumption map of *existence(X, Y)* template.

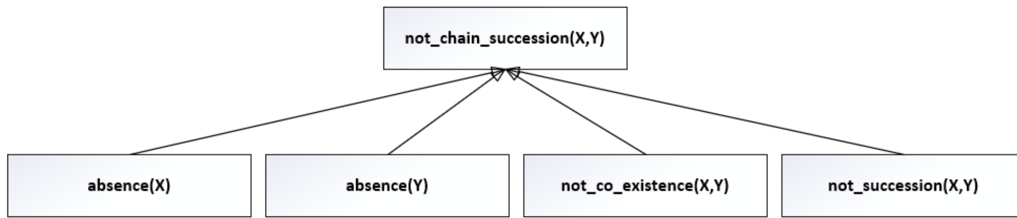

Figure 2.21: Subsumption map of *not_chain_succession(X, Y)* template.

findall function, as will be described in the next section.

In some cases, the specialization of one constraint can not lead to one single specialized constraint, but to two constraints in AND. If this happens, it is mandatory that the two constraints are considered together. For a trace to satisfy that specialization of the original constraint, both of the constraints must hold on it. For example, as visible in Figure 2.18, the co_existence(X,Y) template can be specialized, following

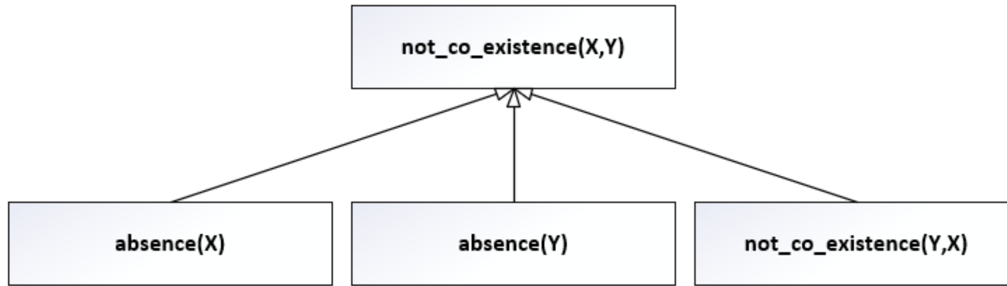

Figure 2.22: Subsumption map of *not_co_existence(X,Y)* template.

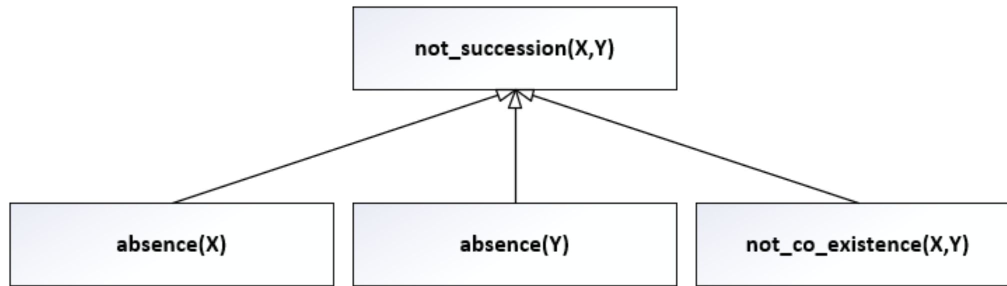

Figure 2.23: Subsumption map of *not_succession(X,Y)* template.

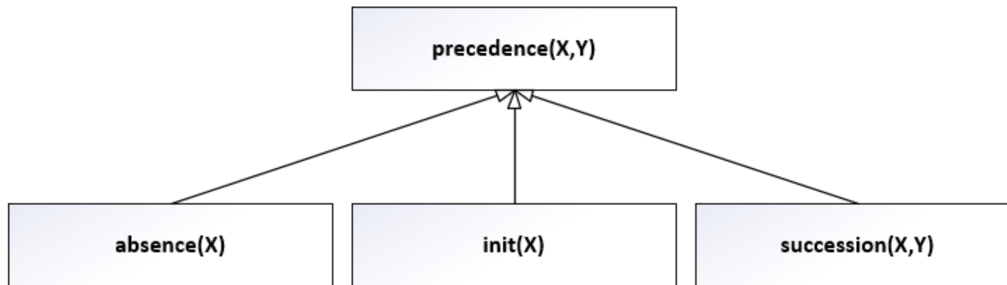

Figure 2.24: Subsumption map of *precedence(X,Y)* template.

this hierarchy, in the pair of constraints responded_existence(X,Y) AND responded_existence(Y,X). For a trace to satisfy this specialization, both of the constraints must be valid on it. This pair of constraint will then be considered as a single one that will therefore not be specialized further. It

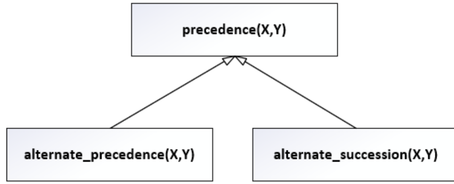

Figure 2.25: Subsumption map of *precedence(X, Y)* template.

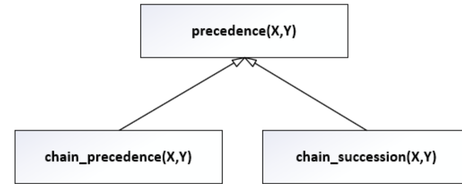

Figure 2.26: Subsumption map of *precedence(X, Y)* template.

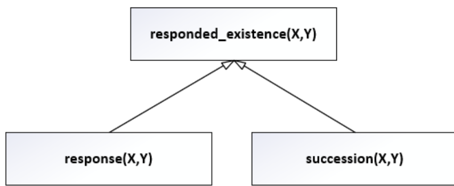

Figure 2.27: Subsumption map of *responded_existence(X, Y)* template.

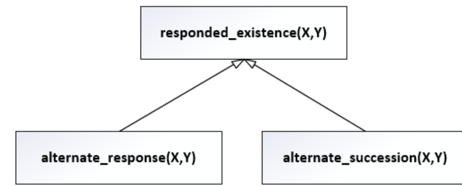

Figure 2.28: Subsumption map of *responded_existence(X, Y)* template.

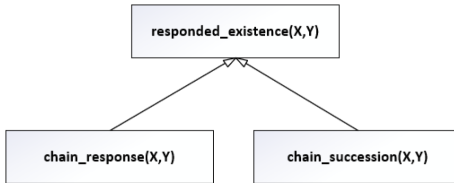

Figure 2.29: Subsumption map of *responded_existence(X, Y)* template.

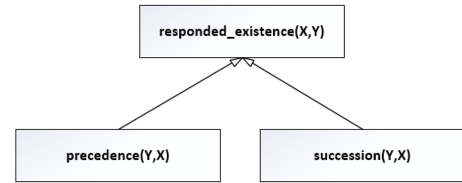

Figure 2.30: Subsumption map of *responded_existence(X, Y)* template.

would be possible though, to specialize the constraints separately, but it would be redundant because both of them are already considered singularly.

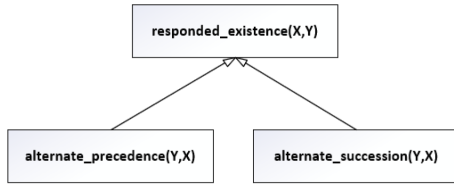

Figure 2.31: Subsumption map of *responded_existence(X, Y)* template.

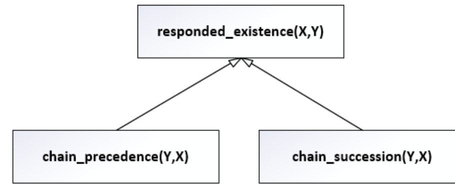

Figure 2.32: Subsumption map of *responded_existence(X, Y)* template.

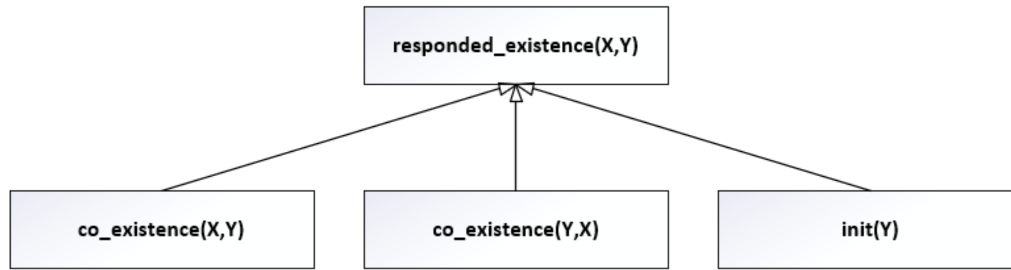

Figure 2.33: Subsumption map of *responded_existence(X, Y)* template.

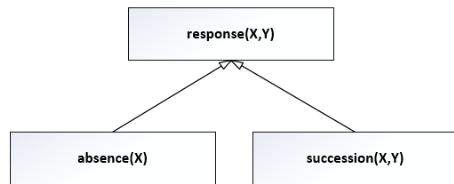

Figure 2.34: Subsumption map of *response(X, Y)* template.

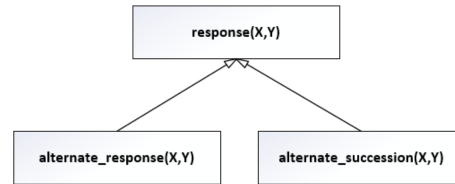

Figure 2.35: Subsumption map of *response(X, Y)* template.

2.3 Choose_Constraint function

The function called in the inner cycle that chooses the next constraint to add to the term is called choose_constraint and it is written in code below.

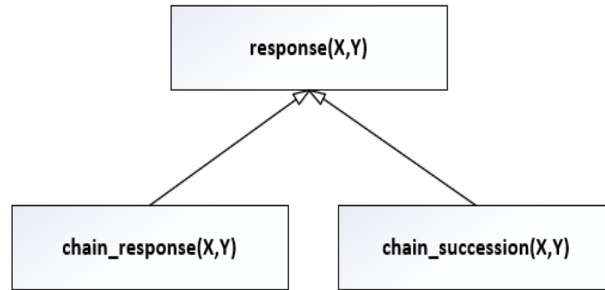

Figure 2.36: Subsumption map of *response(X, Y)* template.

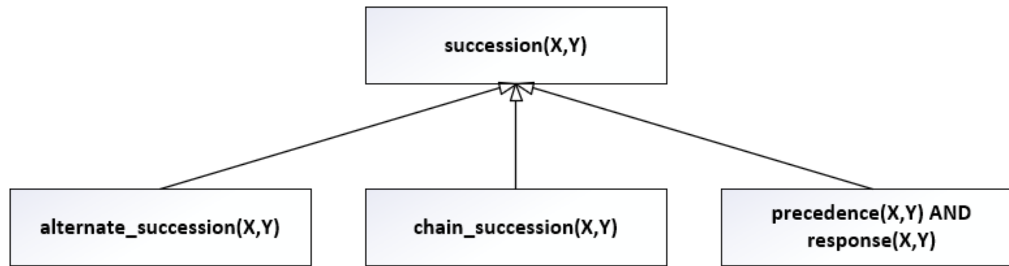

Figure 2.37: Subsumption map of *succession(X, Y)* template.

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

This function has three parameters, the current lists of positive and negative examples and the list of constraints already added to the term, and it returns the constraint that will be added to the term by the inner cycle. The function

needs a list of possible constraints among which it will find the best one, based on the positive and negative traces. To find these constraints, it starts from the first level of the hierarchy and combines the templates with all the possible activities found in the log. For example, if the most general level of the hierarchy only contained the two templates existence(X) and response(X,Y), and the activities found in the log were only a, b and c, then the list of constraints would be:

[existence(a), existence(b), existence(c), response(a,b), response(a,c), response(b,a), response(b,c), response(c,a), response(c,b)]

As the algorithm will present the model in DNF, i.e., every term will be composed of constraints in AND, the first constraints that the function will choose from are the most general ones, that are likely to cover both many positive and many negative traces. In later iterations the function will specialize the model and will have two different ways of doing it: the first one is to choose another constraint, from another branch of the hierarchy, to add in AND to the term; the second one is to specialize one of the existing constraints following the hierarchy.

The reason why the function, differently from the algorithm proposed by Mooney, always starts from the same set of constraints, and not from the ones that already are in the term, is that most of the times many of them would need to be considered anyways because some branches of the hierarchy might not have been explored at all yet, so it is faster to remove the ones that are already present than adding the ones that are not.

The next step consists checking which constraints are already in the term and descending the hierarchy to find the more specialized ones. At the

end of this process the new set of constraints among which to choose in order to find the one that better fits the log is complete and contains the first-level constraints that are not already in the term and all the possible specializations of the ones that were chosen in previous iterations. In this function it is important to consider that, as we always start form the first level, it is not enough to only look for specialization of the starting list of constraints, because the term could already contain one. For example, if in the first iteration the chosen constraint was existence(a) and in the second one existence2(a), in the third iteration the starting list of constraints will still contain existence(a) but looking only for its specializations will not be enough as the term already contains one of them. The function finds the next level of hierarchy and checks which of these new constraints are already in the term. They are removed from the list of constraints that is being built and added to the list that previously contained only the first level of the hierarchy so they can be specialized as well.

In detail, considering the first constraint in the list, if it is already in the term, then all its possible specializations are returned by a findall function called on the hierarchy of templates. Then the constraints that are possible candidates to be added to term and the ones that have already been chosen to be part of it are identified and divided into two variables. The latter list is added to the list of first-level constraints that still need to be checked by the append function and then the function that specializes the existing constraints is called recursively. Please note that here is only described the main part of the function, if the constraint analyzed is not already in term, it is inserted in the list of possible candidates to be added to term; on the other hand, if the constraint is already in term but the findall function fails it is because the constraint has no possible specializations, so it is simply

removed from all the lists.

Once it has the complete list of constraint to choose from, the program removes the duplicates (the same constraint can be the specialization of two different ones) and associates each one with a score, calculated through the dnf_gain function below, called by the get_best function in the pseudocode above, and then chooses the constraint that has the highest one.

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 $p \times (\log_{10}(\frac{p}{p+n}) - \log_{10}(\frac{P}{P+N}))$.

This function, described by Mooney [14], calculates the gain of each constraint taking into consideration the number of positive examples P, the number of negative examples N, the number of satisfied positive examples p and the number of satisfied negative examples n. In our algorithm, if p is 0, the constraint is assigned a very low gain that assures that the constraint will not be chosen. This assignment is necessary because the logarithm of 0 is $-\infty$ and it generates an error in Prolog. Anyways, a constraint that does not cover any positive trace must not be added in the term as, being in AND with all the other constraints, it would make the whole term cover no positive trace and be discarded.

It is important to mention that the algorithm is based on the assumption

that every constraint ever chosen to be part of a term will remain in the term. This is necessary because a constraint in AND with a more specific one becomes useless, but it makes the process of finding the new constraint to add way easier. Starting from the same set of templates and deleting the general constraint every time a more specific one is added would in fact mean that the specialize_existing_constraints function should not only look for the presence in the term of the constraint that it is analyzing, but also for the one of more specific constraints. This variation, analyzed in the next chapter, would make the model simpler but the algorithm quite harder.

2.4 Example

To better understand how the algorithm works, here is an examples on an extremely limited number of traces. Considering our event log to be:

```
Positive traces:

trace(p1,[event(a,1),event(b,2),event(c,3),event(c,4),event(d,5)]).
trace(p2,[event(a,1),event(c,2),event(c,3),event(a,4),event(b,5)]).
trace(p3,[event(a,1),event(e,2),event(c,3),event(a,4),event(b,5)]).
trace(p4,[event(f,1),event(g,2),event(i,3),event(k,4),event(h,5)]).

Negative traces:

trace(n1,[event(a,1),event(k,2),event(g,3),event(f,4),event(e,5)]).
trace(n2,[event(a,1),event(c,2),event(g,3),event(k,4),event(f,5)]).
```

The algorithm starts reading all the positive and negative traces, as the termination condition is that the list of positive traces is empty, i.e. that the model discovered covers all of them, it enters in the outer cycle. The first instruction here is the call to the inner cycle function, that will immediately call the choose_constraint function to find the best constraint to add to the term. This function takes the set of starting templates, extracts all the different activities from the traces and generates the list of constraints to analyze. Then calls the get_best function that will compute the DNF gain of each constraint and select the best one. Here, the first constraint selected is

```
choice(h,b)
```

The choose_constraint function then ends, returning this constraint to the inner cycle. It is now time for the inner cycle to assess how many negative traces do not satisfy this constraint. In this case, the constraint is not valid for any of them, as neither have activities h or b. This means that, thanks to the remove function, the list of negative traces becomes empty and the term, composed only by the constraint choice(h,b), is returned to the outer cycle. The latter now has to check how many positive traces are covered by this term; in this case every positive example has either the activity h or the activity b, so all of the positive traces satisfy the constraint. The remove function removes all the positive traces from the list, that becomes empty and makes the algorithm return the model and end. The model is, in this case, composed of only one constraint, as, by itself, it excludes all the negative examples while covering all the positive ones.

2.5 DNF vs CNF

The CNF algorithm is dual to the DNF one described above. It returns a model in the conjunctive normal form, meaning that it will be an AND of OR. The DNF model was quite easy to understand even in case the log contained positive traces that were completely different from each other as it could describe every different group with a different term. The CNF model is still able to do so, but in the opposite way. If the model has to be described by the same constraints, every clause here will contain one of the constraints that the DNF term contained and there will be as many clauses as the number of constraints in the largest DNF term.

To better understand the differences between the models generated by the two algorithms, if the positive traces in the log are:

trace(p1, [event(a,1), event(b,2), event(c,3), event(d,4)])

trace(p2, [event(a,1), event(b,2), event(b,3), event(c,4)])

trace(p3, [event(a,1), event(c,2), event(b,3), event(d,4)])

trace(p4, [event(k,1), event(c,2), event(a,3), event(d,4)])

And the negative ones:

trace(n1, [event(b,1), event(c,2), event(e,3), event(d,4)])

trace(n2, [event(c,1), event(b,2), event(d,3), event(a,4)])

A process model returned by the DNF algorithm could be:

(existence(a) AND precedence(a,b))

OR

existence(k)

Whereas the CNF algorithm could return a model such as:

(existence(a) OR existence(k))

AND

precedence(a,b)

Note that these two models are not optimal and would not be returned by the algorithm.

2.5.1 Backbone algorithm

As mentioned before, the backbone of this algorithm was inspired by Mooney's CNF Learner. The pseudocode of his algorithm is given below.

The duality between this algorithm and the DNF one is quite easy to see. In this case the outer cycle ends when all the negative traces are excluded by the model and the inner cycle when all the positive ones satisfy the clause. This means that every clause of the algorithm, composed of constraints in OR, will need to necessarily cover all the positive traces in the log. Again, the target language, Declare, does not support the OR, in the algorithm the clauses are represented as lists of constraints just as the DNF terms, and only when it comes to returning the model, it is printed to include it. The inner cycle chooses the most convenient constraint based on a different

Algorithm 3 CNF learner by Mooney

Let Neg be all the negative examples.

Let CNF be empty.

Until Neg is empty do:

  Let Pos be all the positive examples.

  Set Clause to empty and Neg2 to Neg.

  Until Pos is empty do:

    Choose the constraint C that maximizes the CNF-gain.

    Add the constraint C to Clause.

    Remove from Pos all the examples that satisfy C.

    Remove from Neg2 all the examples that satisfy C.

  Add Clause as one clause of CNF

  Remove from Neg all the examples that do not satisfy Clause.

Return CNF.

function that will be described in the next subsection. Then it removes from its local lists of positive and negative examples all the traces that do satisfy it; the DNF algorithm on the other hand, removed the ones that did not. This is because every clause needs to cover all the positive traces, and as every constraint in them is in OR, the examples covered by one constraint are covered by the whole clause too. Removing them from the list makes later iterations focus on the ones that are not yet covered so the clause will not have redundant constraints, that would probably include negative traces too, making the model wider.

The last part of the outer cycle removes form the negative traces the ones that do not satisfy the clause created in the inner cycle and adds it to the model in AND with the other ones. Being in AND, the model is not just as restrictive as the most restrictive clause, which was the reason why every term of the DNF algorithm had to exclude all the negative traces, but every one of them contributes to discard some of them. On the other hand, if a clause does not cover a positive trace, it means that the whole model will not cover it, so the inner cycle only ends when the list of positive examples is empty.

In this version of the algorithm too there is the possibility of excluding some traces, positive or negative, from the model instead of failing, if it is not possible to produce a model that describes them. In this case, the inner cycle takes care of the positive ones, while the outer one removes negatives traces if the inner cycle fails to produce a clause that covers them.

2.5.2 CNF hierarchy of templates

The hierarchy used for the CNF algorithm is the same as the one described above. In this case, though, because the constraints in the clauses are in OR, the set from which the first constraint is selected is not the one containing all the most general templates, as it was for the DNF algorithm, but, on the contrary, is the one made of all the most specialized templates. The hierarchy of templates has to be scanned in reverse; starting from the templates that were the leaves of the tree and possibly reaching the root nodes only in later iterations. The templates in the starting set are:

[absence(X), init(X), end(X), exclusive_choice(X,Y), existence3(X), exactly2(X), chain_succession(X,Y), not_responded_existence(X,Y)]

This set is substantially larger than its DNF correspondent. This is because many of these are subsumed by responded_existence(X,Y), a template that creates a subtree with many different branches.

Because of the way the hierarchy was written, it can remain otherwise untouched, only the findall function used to retrieve the more general constraints will need to be adapted to return the first term of the Prolog fact, SubsumingConstraint, instead of the second one, SubsumedConstraint.

```
next_level_of_hyerarchy(SubsumingConstraint, SubsumedConstraint).
```

2.5.3 CNF choose_constraint function

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

```
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 main difference with respect to the DNF choose_constraint function is the extra combination step. This step 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. Scanning the tree form the leaves to the root, it is then possible that the more specialized constraint only concerned one activity, while the more general one concerns two. For example, as visible in Figure 2.24, both init(X) and absence(X) can be generalized in precedence(X,Y). If the constraint that is already in the clause is init(a), then the more general constraint returned from the function generalize_existing_constraints will be precedence(a,Y) but to check how

many traces satisfy a constraint it is necessary for all of its variables to be grounded. The combine step substitutes the value with every possible activity, creating all the different constraints. Note that in case the template given to the combine function is already grounded it will simply return it as it is.

2.5.4 CNF_gain function

The CNF_gain function is again described by Mooney.

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 $n \times (\log_{10}(\frac{n}{p+n}) - \log_{10}(\frac{N}{P+N}))$.

This function is dual with respect to the DNF_gain one. Here are taken into consideration the numbers of traces, positive and negative, that do not satisfy the constraint, and the number of not satisfied negative examples is the one that is used as numerator in the calculation. This is because the more negative traces are excluded by a clause, the more efficient the algorithm will be. Also, the model will become simpler because the number of constraints in it will be lower. If a constraint satisfies every negative trace, meaning that n is 0, it is assigned a very low gain so it will not be selected.

2.5.5 Example

Again, to better understand how this version of the algorithm works, it is useful to look at an example on a small event log. For simplicity, we will use the same event log used in the example on the DNF version.

```
Positive traces:

trace(p1,[event(a,1),event(b,2),event(c,3),event(c,4),event(d,5)]).
trace(p2,[event(a,1),event(c,2),event(c,3),event(a,4),event(b,5)]).
trace(p3,[event(a,1),event(e,2),event(c,3),event(a,4),event(b,5)]).
trace(p4,[event(f,1),event(g,2),event(i,3),event(k,4),event(h,5)]).

Negative traces:

trace(n1,[event(a,1),event(k,2),event(g,3),event(f,4),event(e,5)]).
trace(n2,[event(a,1),event(c,2),event(g,3),event(k,4),event(f,5)]).
```

The algorithm reads the positive and negative traces and enters in the outer cycle which leads immediately to the inner one. The choose_constraint function takes the set of starting templates, that in this case contains the most specialized ones, and grounds them in the same way described for the DNF version of the algorithm. The get_best function then calculates the CNF gain of every constraint and return the best one, that in this case is

```
chain_succession(f,g)
```

The choose_constraint function, in turn, returns it to the inner cycle, that will now compute how many positive traces are covered by this constraint

and then add it to the clause. In this case, the constraint is satisfied by all of the positive traces because in trace p4 activity g directly follows activity f, while in the other traces the two activities are not present, so the constraint is considered valid. As the list of positive traces becomes empty, the inner cycle ends and returns the clause to the outer one. Here, the negative traces that do not satisfy the clause are removed from the list. Again, all the negative traces are ruled out by the constraint because in both of them activity f is preceded by activity g, not followed. The list of negative traces becomes empty and the algorithm returns the model and ends.

2.6 Optimization

The algorithm, as it was described, is correct, but its execution is not supported by average computers if the number of traces contained in the event log is elevated. An optimization is then needed to make it possible for the algorithm to find a process model regardless of the hardware.

2.6.1 Smaller event log

A first solution would be to use a smaller event log to create the model. This would, on the one hand, make the computation lighter as it would decrease the RAM usage and avoid the “Out of stack” error, i.e., the necessity of the algorithm to use more memory than available. On the other hand, it is possible that the model would be less precise, perhaps not including some correct trace, or not excluding negative ones, that would otherwise have been in the event log. As this solution could produce incorrect models it should not be implemented.

2.6.2 Analyzing only a subset of constraints

A different approach would be to stop the analysis of the constraints as soon as the algorithm finds a "good enough" one. This would require a threshold that represents the minimum gain that a constraint should have in order to be added to the term or clause. The first constraint found that has a gain higher that this threshold would be chosen, and the other ones would not be analyzed at all, making the computation extremely faster. The threshold, though, would have to decrease at every iteration of the outer cycle because, as the number of positive traces decreases, the number of covered positive traces will reduce as well, and it may not be possible to find any constraint with a high enough gain.

The choice of the starting value for the threshold heavily depends on the dataset, so the best way to set it is to first set a very high one, then, based on the gain of the first constraints analyzed, choose an appropriate value. In the outer cycle the value can then decrease by a fixed quantity or based on the number of traces excluded by the new term or clause. The drawback of this solution is that the generated model is not going to be optimal, meaning that the algorithm will not find the optimal one. For example, the model found by the algorithm used this way is:

```
(existence(reject_application) AND
 existence(receive_negative_feedback) AND
 existence(approve_application) AND
 not_chain_precedence(approve_application,notify_approval))

OR
```

Algorithm 4 Modifications to the original algorithm

▷ The assignment of the Threshold value would happen in the outer cycle

▷ The length of the term is equal to the number of iterations of the inner cycle

```
length(Term, NumberOfIterations),
Threshold is StartValue - DecreaseValue/NumberOfIterations,
```

▷ The get_best function would return a constraint as soon as it founds one that has a gain higher than the threshold

get_best([Constraint | ListOfPossibleCandidates], ListOfPositiveTraces,
ListOfNegativeTraces, Threshold, NewConstraint) :-
  dnf_gain(Constraint, ListOfPositiveTraces, ListOfNegativeTraces,
   Gain),
  If Gain ≥ Threshold,
  NewConstraint is Constraint.

```
(existence(receive_positive_feedback) AND
 existence(notify_approval))
```

Whereas the one found on the same dataset but analyzing every constraint is:

```
(exclusive_choice(send_acceptance_pack,
  receive_negative_feedback)
 AND
 choice(send_acceptance_pack, receive_negative_feedback))
```

Both of these models are correct, but the second one is clearly simpler and uses different constraints, whereas the model found by stopping the analysis mostly uses existence constraints because, being in alphabetical order, they are the ones that are analyzed first.

2.6.3 Assert and Retract predicates

The last analyzed technique is to use the Prolog predicates assert and retract; they allow to store facts into the heap, therefore freeing the stack. Instead of having the list of combined constraints stored in a variable, every constraint is asserted and then they are retracted in smaller groups to be analyzed. The asserted constraints need to be retracted based on the template, the algorithm gets a list that contains all the templates and then retracts every constraint. The result will be the first constraint found in the heap that matches it, for example existence(a). In order to also find the other two constraints (existence(b) and existence(c)) the retract

predicate above must be executed two more times. Note that if this is executed in a recursive call, it will not be possible to use the same variable for every execution because the retract predicate will bind the variable in the template. A simple solution is to make a copy of the template using the predicate copy_term and using it to retract the constraint needed, leaving the original template untouched.

Assert and retract can also be used to implement the tabling technique [15]. To calculate the gain, it is necessary to know how many traces satisfy the constraint. This means that every trace is tested every time a constraint is analyzed, but after the first call, we already know if a constraint is valid on it, so it does not make sense to test it again. It is possible to assert a fact that will avoid these useless calls to the verify_constraint function on the traces that were already tested. It will look like

```
assert(verified(TraceID, Constraint, Answer)).
```

The variable Answer can only assume two values, "yes" or "no"; if the value is "yes", it means that the constraint is valid on that trace, on the other hand, if the value is "no", it means that the trace does not satisfy the constraint. Note that, in this case, the predicate retract is not used because the constraint remains valid or not valid on a trace for the whole computation, whereas the list of constraints to analyze changes at every iteration. In order to get the answer it is enough to call

```
verified(TraceID, Constraint, Answer)
```

With the variables TraceID and Constraint grounded. If the fact was asserted, this will ground the variable Answer and return the value that we are looking for.

This technique is based on the assumption that every trace has a different identifier, if, for example, both positive and negative trace identifiers are numbers that start form 1 for the first trace and increase for the following ones, then there will be a trace with ID = 1 among the positives and another one among the negatives. This would lead to errors because it is possible that a constraint is valid for the positive trace with ID = 1 but not for the negative one. The assert predicate called after the function verify_constraint on the positive trace would then create a fact that would be found when checking the constraint's validity on the negative trace with the same ID. Moreover, this is only useful to speed up the computation from the second iteration of the inner cycle, as for the first one all the constraints have never been analyzed before. The drawback is that it slows down the first iteration a little.

Another observation that can be made is that if a constraint does not hold on a trace, also more specified ones will not. For example, if the constraint existence(a) does not hold on trace 1, then also exacly1(a), exactly2(a), existence2(a), existence3(a) and init(a) will not hold on it. It would be then possible, when we find that a constraint does not hold on a trace, to get all the specialized ones and assert them as well as not valid. The same reasoning can be applied the other way around in the CNF version of the algorithm, if we find that a specialized constraint holds, then also all the more general ones will. In practice, though, this has the drawback of slowing down the computation during the first iteration of the inner cycle, that was already

the slowest one. This is due to the fact that scanning the hierarchy every time to find the other constraints to assert requires computational effort as it is not enough to find all the immediate specializations of a template, the hierarchy has to be analyzed till the leaf nodes. It is also possible that this process generates doubles in the list, for example, the existence constraint already has exactly2(X) and existence3(X) among its possible specializations, but they are also the ones of existence2(X), so scanning the whole hierarchy starting from existence(X) would produce a list with those templates repeated twice. All in all, this is very useful if there are multiple iterations of the inner cycle in the algorithm because even if the first one becomes slower, the following ones will be extremely faster. As the models that can be defined by only one constraint are quite rare, this technique was implemented in the algorithm.

Chapter 3

Experimental results

The algorithm was tested on different event logs with the aim of, firstly, assess whether it could find a correct process model and, secondly, analyzing its performances. The process model of the first event log used was already known, testing the algorithm on it allowed to confirm the correctness of the discovered model. All the datasets were useful for the performance analysis; the three different properties taken into consideration were the time taken by the algorithm to return the final model and the occupation of the global and local stack.

3.1 Controlled event log

The two algorithms were initially tried on a controlled event log where the traces were artificially generated from a known process model, described in figure 3.1 and taken form [13], so that they could be excluded by a single constraint. The main focus will, in

fact, be on the three constraints exclusive_choice(send_acceptance_pack, receive_negative_feedback), visible in the bottom part of the figure, precedence(appraise_property, asses_loan_risk), at the top, and precedence(assess_loan_risk, assess_eligibility), that are valid for every positive trace, but are the ones that the three different sets of negative examples violate.

The log contains 64000 positive traces and has three different sets of negative ones. The activities for this log are 16 and are listed below:

- appraise_property;
- approve_application;
- ask_for_customer_feedback;
- assess_eligibility;
- assess_loan_risk;
- cancel_application;
- check_credit_history;
- check_income_sources;
- notify_approval;
- notify_cancellation;
- receive_loan_application;
- receive_negative_feedback;
- receive_positive_feedback;

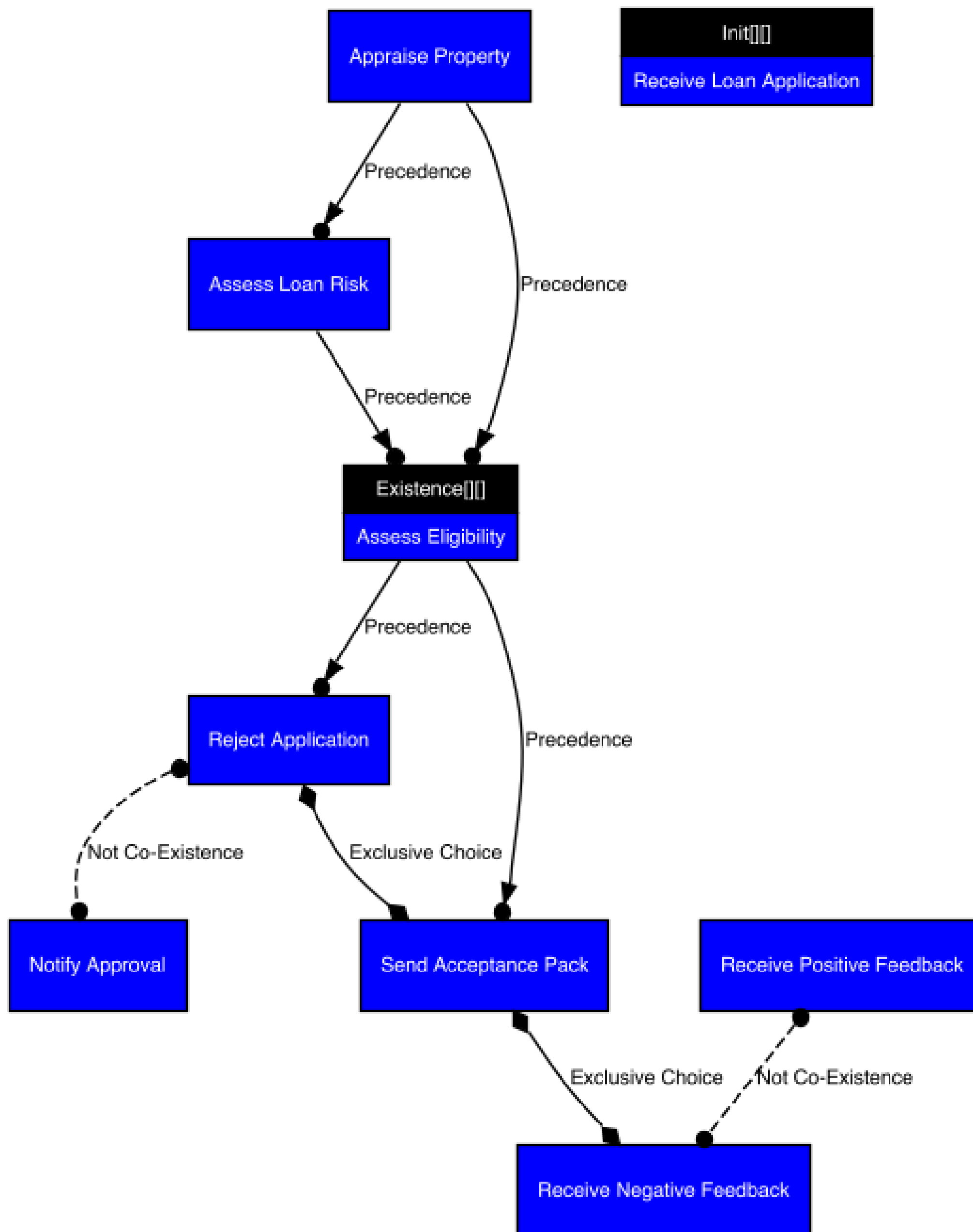

Figure 3.1: Process model from which the positive and negative traces were generated

- reject_application;
- send_acceptance_pack;
- verify_receipt.

3.1.1 First set of negative traces

The first set contains 10240 negative examples and all of them can be ruled out by the constraint exclusive_choice(send_acceptance_pack, receive_negative_feedback). The DNF version of the algorithm returned the following model:

```
(exclusive_choice(send_acceptance_pack,
  receive_negative_feedback)
 AND
 choice(send_acceptance_pack, receive_negative_feedback))
```

The reason of the presence of the constraint choice(send_acceptance_pack, receive_negative_feedback) in the model is that the algorithm starts analyzing only the most general constraints. Being exclusive_choice(send_acceptance_pack, receive_negative_feedback) a specialization of the previous constraint, it was not analyzed during the first iteration of the inner cycle, but only on the second, after the constraint choice(send_acceptance_pack, receive_negative_feedback) was chosen and its specializations were included in the list of constraints to analyze. The CNF version of the algorithm, on the other hand, starting from the most specialized constraints and generalizing in the following

iterations, returned the model constituted only of expected constraint:

```
exclusive_choice(send_acceptance_pack, receive_negative_feedback)
```

Form a performance point of view, having to iterate only once, the CNF algorithm was expected to be faster. After a running the two algorithms a few times, it was possible to retrieve more accurate statistics. the time taken by the CNF algorithm to return the model was around 297 seconds (the algorithm always returns the same model but the time that it takes can vary of a few seconds depending on what other thing the system is doing while the algorithm is running; for example, I/O operations can slow it down a little), whereas the DNF one took about 1129 seconds. The CNF algorithm was around four times faster, even though it starts with a higher number of templates in the initial set. The reason for this is that the template not_chain_succession(X,Y), that is in the starting set of the DNF algorithm, turned out to be harder to compute that the other ones, therefore slowing down the whole algorithm. It is also important to consider that the starting set of templates of the DNF algorithm contains a total of five templates, three of which with two variables and two with one variable, while the CNF one contains a total of eight templates, but three have two variables and the other five have only one. As the templates with one variable create 16 constraints (the same as the number of activities) and the ones with two create up to 240 constraints, the higher number of single-variable templates does not make a great difference in the performance. Note that the two variables must have different values, so all the constraint like response(a,a) can be discarded.

3.1.2 Second set of negative traces

The second log contains 12800 negative traces that can be ruled out by the constraint precedence(appraise_property, assess_loan_risk). Both the DNF and the CNF model, though, were quite different form this as the DNF one was:

```
not_chain_succession(assess_loan_risk, appraise_property)
```

And the CNF one was:

```
chain_succession(receive_loan_application, appraise_property)
```

After checking the event log, both these models were verified to be correct and they were found in a relatively short amount of time, as both the algorithms had to only find one constraint and therefore only go through one iteration of the inner cycle. The performance differences given by the presence of the template not_chain_succession(X,Y) in the DNF version are again striking, the CNF algorithm found the model in about 307 seconds, while the DNF one found it in around 700 seconds, so the CNF was about twice as fast.

The fact that the constraint precedence(appraise_property, assess_loan_risk) is not in any of the two models is easily explained because both algorithms have found a correct model using only one constraint, that belongs to the first set of constraints that were analyzed. On the other hand, the template precedence(X,Y) is neither in the starting set of the DNF algorithm nor in the one of the CNF algorithm.

3.1.3 Third set of negative traces

The third negative log contains 25600 traces, and it is made so that all of them can be ruled out by the constraint precedence(assess_loan_risk, assess_eligibility). Having way more traces than the previous ones and not having the constraint in the first set of analyzed ones, the algorithm was expected to take a longer time to extract the model. Once it finished though the DNF model was:

```
(not_chain_succession(assess_loan_risk, appraise_property)

 AND

 not_chain_succession(assess_eligibility, assess_loan_risk)

 AND

 existence(assess_loan_risk))
```

The first constraint returned was existence(assess_loan_risk) that ruled out half of the negative examples, the second one was not_chain_succession(assess_eligibility, assess_loan_risk) that ruled all the remaining negative traces but one and, finally, the last one was not_chain_succession(assess_loan_risk, appraise_property) that ruled out the remaining negative trace. Luckily, this term supports all the positive traces.

The model returned by the CNF algorithm is, instead, compliant with the expected one:

```
(precedence(assess_loan_risk, assess_eligibility)

 OR
```

```
chain_succession(assess_loan_risk, assess_eligibility))
```

The first constraint found, chain_succession(assess_loan_risk, assess_eligibility)), is valid on 63872 positive traces, the second one, precedence(assess_loan_risk, assess_eligibility), that was introduced in the set of constraint to analyze as a specialization of the previous one, is the one that was expected, so it covers all the positive traces. Both of them rule out all the negative traces, so no other clause was needed.

Again, these two models are correct, and the reason why the expected constraint is not in the DNF model is to be found in the starting set of templates. If the algorithm starts with a starting set that contains the constraint precedence(X,Y), then returned the model is, as expected, precedence(assess_loan_risk, assess_eligibility).

The performance of the CNF algorithm was, as anticipated, much better that the one of the DNF. The former returned the model after an average of 522 seconds, whereas the latter took about 1648 seconds. Here the DNF version of the algorithm took around three times the time of the CNF.

3.1.4 Considerations on the performance

After also considering the previous tests, it is possible to conclude that, for this controlled event log, the DNF will take about twice as long as the CNF to choose a single constraint. In the first test the number of constraints in the DNF model was 2 whereas the one in the CNF model was 1, in the second test the two numbers were both 1, and here they were, respectively, 3 and 2. As the CNF model was found respectively about four times, two times and three times faster than the DNF one, it is easy to see how for

	global stack	local stack
first test DNF	42506608	3528
first test CNF	77821144	2568
second test DNF	50136944	3040
second test CNF	80386488	2568
third test DNF	92748272	4016
third test CNF	107962248	2568

Table 3.1: Stack occupation after the execution of the algorithm

every constraint by the DNF algorithm, the CNF on can find two.

Calling the predicate statistics(stacks, X) at the beginning and at the end of the execution of the algorithm, it is possible to see the memory in use divided in global and local stack. In all the tests, the result before the algorithm started was the same for both the CNF and the DNF algorithms:

[11944, 1560]

These are respectively the values that indicate the usage of the global and local stack. In table 3.1 are listed the stack occupation recorded right before the termination of the algorithm, after the model was found. They are also visible in the graphs in figures 3.2 and 3.3.

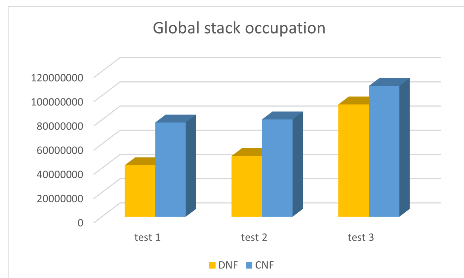

Figure 3.2: Subsumption map of *co_existence(X, Y)* template.

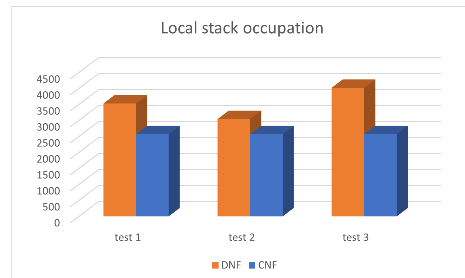

Figure 3.3: Subsumption map of *existence2(X, Y)* template.

These results reflect the different number of traces in the different tests. It is evident how the third one, that has the highest number of negative examples, had to store more facts using the assert predicate with respect to the other two tests.

3.2 Pap Test screening event log

Another test was run on a different event log [16] that contains traces relative to a pap test screening, that aims to detect the presence of the papillomavirus infection in women over the age of 25. The number of activities was a little higher than the one in the previous tests as they were:

- 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_negative_biopsy;
- 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.

In contrast, the total number of traces was tremendously lower as there were only 55 positive traces and 102 negative ones. This was reflected in the time the two versions of the algorithm took to return the model; the CNF one was again faster with about 3 seconds but this time the DNF was quite fast too, giving the result after a little less than 5 seconds. The two models were respectively:

```
(exclusive_choice(send_letter_negative_biopsy,
    send_result_inadequate_papTest)
  OR
  chain_succession(phone_call_positive_papTest,
    execute_colposcopy_exam))
AND
```

	global stack	local stack
DNF	174888	3336
CNF	142080	4768

Table 3.2: Stack occupation after the execution of the algorithm

```
(chain_succession(invite, refuse) OR chain_succession(invite,
    execute_papTest_exam))
```

And

```
choice(refuse, send_result_inadequate_papTest)
OR
(exactly1(send_letter_negative_papTest)
  AND
  choice(send_letter_negative_papTest, execute_colposcopy_exam))
```

The memory occupied by the algorithm reflects the fact that the traces are less that the ones in the previous test, in able 3.2 are the two outputs of the statistics function called with the parameter stacks for DNF and CNF.

Note that in this case, the output before the execution was:

[12960, 1672]

3.3 Real-life event log with no negative traces

The final test was run on a real-life event log that contains events of sepsis cases, a life-threatening condition typically caused by infection, from a hospital [17]. Everyone of the 1050 traces represents the pathway through the hospital. The different activities are 16:

- admission_ic;
- admission_nc;
- crp;
- er_registration;
- er_sepsis_triage;
- er_triage;
- iv_antibiotics;
- iv_liquid;
- lactic_acid;
- leucocytes;
- release_a;
- release_b;
- release_c;

- release_d;
- release_e;
- return_er.

This event log also recorded data attributes that could be used to extract other useful information, see the next chapter for a full discussion on this topic.

The peculiarity of this event log with respect to the previous ones is that all the 1050 traces that it contains are to be considered positive. The two algorithms described in the previous chapter are not suitable for this: the CNF version's termination condition is that the list of negative traces becomes empty so it would end without returning any model; the DNF version would instead consider the model impossible to return because the inner cycle's termination condition is that the list of negative traces is empty, so it would return an empty list instead of a term and, as the empty list does not cover any positive trace, they would all be considered impossible to cover by any constraint. For this reason, the algorithm was extended to be able to operate also when there are no negative examples. The DNF version, starting form the most general constraints, it is bound to return a model composed by very few constraints, all in OR as there is no negative trace to exclude, whereas the CNF version will probably need to insert more constraints in the model, again in OR, because it starts from the most specialized constraints, that will probably not cover a high number of traces.

This event log, containing a small number of examples, made it possible for the algorithm to return very easy models. The DNF one only has one constraint:

```
absence3(admission_ic)
```

While the CNF model has two:

```
absence2(release_e) OR absence(release_e)
```

Note that in the latter the first constraint found, absence(release_e), covered all the positive traces but 6, which are covered by the second one. Besides, the reason why the models are only composed of absence constraints is that the sorting operation that removes the doubles also ordinates the constraints in alphabetical order, so even though there were other constraints that covered the same number of traces, the first one found were, in both cases, the ones generated from the absence templates.

The time taken by the DNF version is double the one taken by the CNF one, they are respectively about 14 and 7 seconds. The time taken by the CNF algorithm to find the second constraint is less than 1 second, the very low number of traces left and the fact that the function verify_constraint asserts the results for the already analyzed traces make the second iteration computationally much easier than usual. On the other hand, the DNF algorithm is again slowed down by the computation of the constraints derived from the not_chain_succession template.

The values for the stack again started from:

[12960, 1672]

At the end of the execution, they became the ones listed in table 3.3.

	global stack	local stack
DNF	2008976	2064
CNF	1954464	2064

Table 3.3: Stack occupation after the execution of the algorithm

3.4 Final considerations

The complexity of the discovered model heavily depends on the event log. The models returned by the CNF algorithm were simpler in the case of the Pap test event log and of the first and third tests on the controlled event log. On the other hand, the model given by the DNF version in the test on the sepsis event log was the simpler one. On the second test on the controlled event log both versions returned a model composed only of one constraint. The reason why the two algorithms return different models lays in their starting sets of templates. The sets of negative traces in the first and third tests were made to be ruled out by a constraint that was respectively already in the starting set of the CNF algorithm and a direct specialization of the constraint with the highest gain in its starting set, whereas the path to reach it was longer following the hierarchy from the DNF's starting set. In contrast, the model discovered using only positive traces, such as the Sepsis one, will always be simpler if discovered by the DNF version of the algorithm, even though it will be way more general than the CNF's one.

In table 3.4 are given the temporal performance data for both the CNF and the DNF versions of the algorithm. The column on the right emphasizes the number of positive and negative traces in the different event logs, the first addend represents the positive ones and the second addend the negative

Dataset	DNF	CNF	Total number of traces
First test on controlled event log	1129s	297s	64000 + 10240
Second test on controlled event log	700s	307s	64000 + 12800
Third test on controlled event log	1648s	522s	64000 + 25600
Pap test event log	5s	3s	55 + 102
Sepsis event log	14s	7s	1050

Table 3.4: Temporal performance

ones; note that the last dataset did not have any negative trace and the number in the table corresponds to the positive ones.

The number of traces, as expected, highly influences the overall time taken to return the model. Nevertheless, the CNF version of the algorithm returned the process model considerably faster for every event log. As previously stated, this was not predicted because the starting set of templates of the CNF version is larger than the DNF version's one, but the tests made clear that the bottle neck of the latter is the analysis of the constraints that come from the not_chain_succession(X,Y) template. The part that most influences the temporal performance of the algorithm was observed to be, as it was anticipated, the first iteration of the inner cycle, because all the constraints need to be tested on every trace. Later iterations are faster thanks to the asserted facts that state whether a certain constraint is valid on a trace or not.

The stack occupation too depends on the number of traces in the event log. It is also important to note that it is also influenced by the number of possible specializations of the constraints that are added to the term at each iteration of the inner cycle. Constraints that are more general will, in fact, usually have more specializations that the already specialized ones.

The focal point, though, is that thanks to the optimization described above, the stack occupation never exceeds the limit and the algorithm is able to run also on average computers.

Chapter 4

Conclusions and future work

In this work a declarative process mining algorithm was presented. It was inspired by the one written by Mooney [14] and aimed at discovering the model of a process that will be described by Declare constraints connected by logical operators. It is crucial to remember that the focus of this thesis is the utilization of event logs that contain both positive and negative traces, whereas the majority of the existing process mining algorithms compute the process model taking in input only positive examples. Negative traces can lead to the discovery of easier and more accurate process models because they allow to analyze the reasons why some traces deviate from the path that leads to the successful completion of the process. At first the technologies that were used in the project were discussed in detail as to make the implementation of the algorithm clear for the reader. Secondly, the algorithm's implementation was thoroughly explained, with a particular focus on the two different versions and on the relations among the several Declare templates that are used to define the process model. The most important part of the algorithm is in fact the selection of the correct

constraint to add to the model, which is done differently in two versions: the DNF one starts from the most general constraints and then specializes them in later iterations to return a model in disjunctive normal form, whereas the CNF one starts from the most specialized constraints and then generalizes them to return a model in conjunctive normal form. The comparison between the two algorithms is carried on in the third chapter, where the process models found by the algorithm for different event logs are examined.

Taking into consideration the test done on the sepsis cases event log, where there were no negative examples, the DNF version of the algorithm returned a model composed by one of the constraints that covered all the positive traces. The algorithm could have chosen any of these constraints because, as they all covered all the traces in the event log, they all represented a correct process model. Having negative traces to analyze would probably have ruled out a few of these constraints because some of them would have covered some negative examples too. This would have led to a more precise process model.

In the second chapter, the optimizations done to make the algorithm run also on average computers are described. Obviously, there is more that can be done to improve the algorithm both in terms of performance and of functionalities. For example, as mentioned above, some event logs record attributes for every activity that could be useful to discover other information. Some of these additions are discussed in the next section.

4.1 Future work

4.1.1 Optimization of the returned model

The models returned by this algorithm contains both the original and the specialized, or generalized, constraint. The original constraint, though, is not relevant anymore because it is completely overwritten by the specialized, or generalized, one. The choose_constraint function could be modified to return both the new constraint added to the term, or to the clause in the CNF version of the algorithm, and the new term or clause, that will be referred to as list of constraints. In this case the function would become:

```
choose_constraint(ListOfPosEx, ListOfNegEx,
  ListOfConstraintsInAND, NewListOfConstraints, NewConstraint)
```

The new list would simply contain the old constraints and the new one or, in case of a specialization or generalization, would substitute the old constraint with the new one and return the list with the same number of elements as before. For example, if the constraint *existence(a)* is specialized in *init(a)* than the presence of *existence(a)* in the list becomes irrelevant, as the new constraint is more restrictive. Removing the constraints from the list would make it possible to generate simpler models, but it would be harder to identify further specializations, or generalizations, in later iterations. As already stated, the choose_constraint function with four parameters is however still correct, but it is does not return the perfect model.

For example, the algorithm could return the following model:

(**existence(a)** AND response(b,c) AND **init(a)**) OR

response(a,d)

The first term contains both the constraint *existence(a)* and its specialization *init(a)*. The choose_constraint function described above would instead remove the general constraint and return the model:

(response(b,c) AND init(a)) OR

response(a,d)

A different way to solve this problem would be to keep the choose_constraint function as it is and then take out, from every term or clause, the constraints that are useless because they have been specialized or generalized. This solution would be simpler because the code would be left essentially untouched, so there would not be the problem of finding the specializations, or generalizations, of already specialized, or generalized, constraints, and it would at the same time return the simplest model possible.

4.1.2 Properties of the activities

The traces considered in this project are in the form:

trace(TraceID, [event(ActivityName, Timestamp), . . . ,
  event(ActivityNameN, TimestampN)]).

The only two things that are taken into account are the name of the

activity and its timestamp. It is possible though that in the event log every activity is associated with more information. For example, instead of just ActivityName, the activity could be expressed as:

activity(ActivityName, _, _, . . .)

And the traces would become:

trace(TraceID, [event(activity(ActivityName, _, _, . . .), Timestamp), . . .]).

Where the underscores indicate other properties of the activity. For example, if the activity is a login on a website, then the other terms could contain the username and the password of the user. If the username is john and the password is 1234, the activity becomes:

activity(login, john, 1234)

This new information could be useful for future developments, it would make it possible to perform other tasks, different from the generation of the model; for instance, the user could perform a statistical research on how many times a person logs in the website or how often people type the wrong password. The latter could lead to identifying attempts of hacking a profile.

The drawback is that not every event log contains all this information; in order to have correct conclusions, the data has to be complete and correct for every activity. As described in Section 1.1.2, the event log should be treated as a first-class citizen, otherwise every model that is extracted from it will not reflect reality.

4.1.3 Unbounded variables in the model

The model returned by the algorithm could contain also templates instead of only constraints. For example, *response(X,Y)*, returned in the final model, would simply mean that there must be at least two activities in every trace. Here the variables could be existentially quantified or universally quantified using a range restriction. Being able to choose templates to add to a term would lead to different specializations. Other than the ones given by the hierarchy, it would be possible to specialize the template by grounding its variables. The template *response(X,Y)*, having two unbounded variables, would then lead to three new different kinds of specializations: the first one given by the grounding of the variable X, the second one of the variable Y and the third one of both variables. This means that even having only two possible activities a and b, the number of new constraints to analyze would be quite high, as can be seen in Figure 4.1.

It is possible, though, that the models returned using this strategy would be significantly simpler for some event logs, especially if combined with one of the solutions in 4.1.1.

4.1.4 Using CLP

It could be possible to use constraint logic programming (CLP) [18] both for the implementation of the Prolog algorithm and for the processing of the Declare variables. The variables would be defined on a finite domain rather than being either bound, i.e., having a value, or unbound. Furthermore, the programmer can specify rules that restrict the domain of the variables. This makes it possible to solve complex problem with a minimum amount of code.

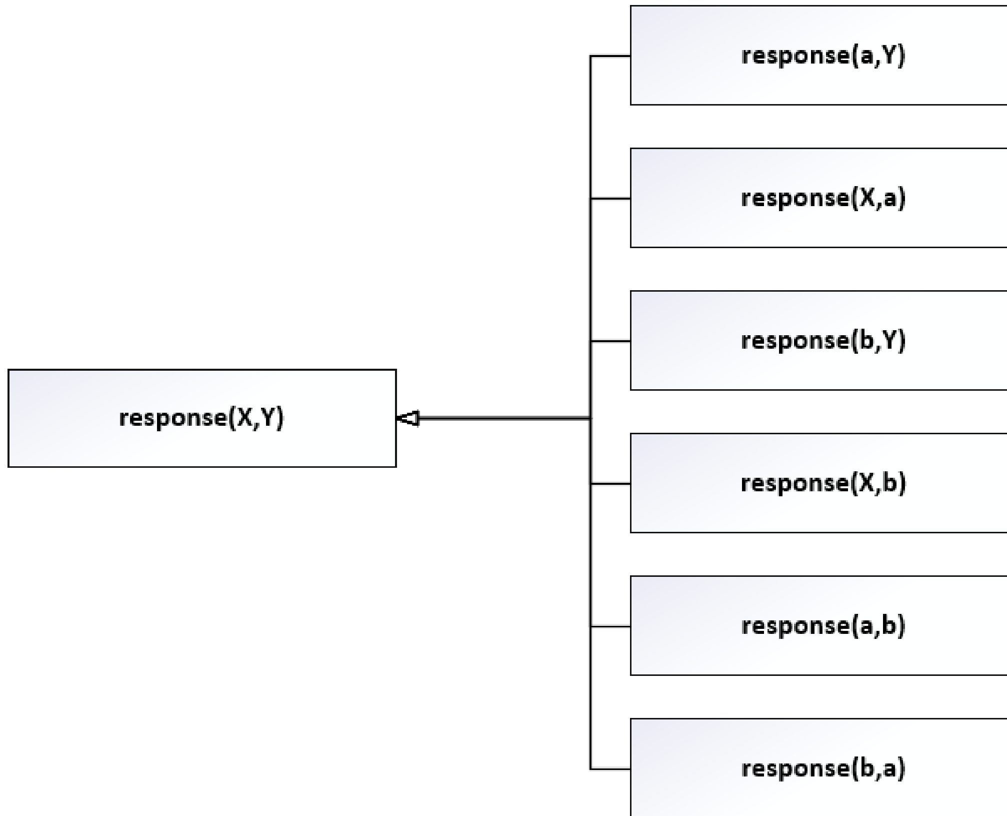

Figure 4.1: New specializations of the template *response(X, Y)*

For example, we could say that the variable X can only take on integer values from 1 to 5, that, through an enumeration, can correspond to the activities a, b, c, d, and e. Saying that the variables of a response constraint must be different from each other would be quite easier, as it would be enough to state that for the constraint response(X1, X2), X1 must be different from X2. The same reasoning can be applied to say that for the exclusive_choice constraint, exclusive_choice(X1, X2) is equivalent to exclusive_choice(X2, X1). This way the verification instructions can be thought of as some sort of constraint.

4.1.5 Asserting traces

A further optimization that can be made using the assert predicate consists in asserting every trace. This would allow to get rid of the variables that contain them and, instead of verifying a constraint on every trace using a recursive call, the same result could be obtained through a simple findall function. The latter would call automatically the verify_constraint function and return the list of traces for which the constraint is valid, therefore the number of traces needed in the dnf_gain function would simply be calculated as the length of that list.

4.1.6 Selecting preferred templates

Right now the DNF gain formula is:

$$p \times (log_{10}\frac{p}{p+n} - log_{10}\frac{P}{P+N})$$

The value that it returns is strictly based on the number of positive and negative traces covered by the constraint in relation to their total number. It could be possible to assign a "score boost" to the templates that the user prefers to have in the model. For example, if the user wants the model to contain mostly responded_existence constraints, we could state that the gain of the responded_existence constraints is the one given by the formula above, plus 100. This way, if one of them would have had the same score as, for example, a choice constraint, we are sure that the choose_constraint function will return the one that we are most interested in. Different templates could have a different score boost that would help the algorithm structure the model in the way requested by the user.

$$(p \times (log_{10}\frac{p}{p+n} - log_{10}\frac{P}{P+N})) + ScoreBoost$$

The drawback would be that the returned model may not be the optimal one, because the score boot could make the algorithm select a constraint that would have had a lower DNF score, meaning that it covers less positive traces or excludes less negative traces with respect to the constraint that would have hat the highest DNF gain.

Bibliography

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

[2] M. Pesic and W.M.P. van der Aalst, 2006, *A Declarative Approach for Flexible Business Processes Management* https://www.researchgate.net/publication/221585980_A_Declarative_Approach_for_Flexible_Business_Processes_Management

[3] Declarative process modeling and mining http://www.diag.uniroma1.it/~marrella/slides/pm17/LAB-07-Declarative_Process_Modeling_and_Mining.pdf

[4] Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, Jan Mendling, 2016, *Resolving inconsistencies and redundancies in declarative process models*

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

[6] Pesic, M. (2008). *Constraint-based workflow management systems: shifting control to users.* Technische Universiteit Eindhoven. https://doi.org/10.6100/IR638413

[7] 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

[8] Lloyd, J. W. (1987). *Foundations of Logic Programming. (2nd edition).* Springer-Verlag.

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

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

[11] *SWI-Prolog*, https://www.swi-prolog.org/

[12] Lamma E., Mello P., Riguzzi F., Storari S. (2008) *Applying Inductive Logic Programming to Process Mining.* In: Blockeel H., Ramon J., Shavlik J., Tadepalli P. (eds) Inductive Logic Programming. ILP 2007. Lecture Notes in Computer Science, vol 4894. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78469-2_16

[13] Federico Chesani, Chiara Di Francescomarino, Chiara Ghidini, Daniela Loreti, Fabrizio Maria Maggi, Paola Mello, Marco Montali, Sergio

Tessaris, *Process discovery on deviant traces and other stranger things*, To be submitted.

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

[15] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan and Jia-Huai You, 2002, *Implementation of a Linear Tabling Mechanism* https://www.researchgate.net/publication/2862749_Implementation_of_a_Linear_Tabling_Mechanism

[16] Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari, 2007, *Inducing Declarative Logic-Based Models from Labeled Traces* https://www.researchgate.net/publication/221586322_Inducing_Declarative_Logic-Based_Models_from_Labeled_Traces

[17] Sepsis Cases - Event log https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639/1

[18] Constraint Logic Programming in Prolog https://en.wikibooks.org/wiki/Prolog/Constraint_Logic_Programming