Vinícius Michel Gottin

Discovery, Conformance and Enhancement of Educational Processes via Typical Plans

Tese de Doutorado

Thesis presented to the Programa de Pós-Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Antonio Luz Furtado

Rio de Janeiro
November 2019

Vinícius Michel Gottin

Discovery, Conformance and Enhancement of Educational Processes via Typical Plans

Thesis presented to the Programa de Pós-Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the Examination Committee.

Rio de Janeiro, November 26th, 2019

Vinícius Michel Gottin

Obtained a B.Sc. in Social Communication - Marketing and Advertisement at Universidade Federal de Santa Maria (UFSM) in 2008. Obtained a B.Sc. in Computer Science at Universidade Federal de Santa Maria (UFSM) in 2011. Received his M.Sc. degree in Information Systems from Universidade Federal do Estado do Rio de Janeiro (UNIRIO) in 2013. Joined the Doctorate program in Informatics at PUC-Rio in 2015, researching conceptual modeling, automated planning and process mining.

Bibliographic data

Gottin, Vinícius Michel

Discovery, conformance and enhancement of educational processes via typical plans / Vinícius Michel Gottin ; advisor: Antonio Luz Furtado. – 2019.
209 f. : il. color. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2019.
Inclui bibliografia

1. Informática – Teses. 2. Mineração de processos. 3. Planejamento automatizado. 4. Modelagem conceitual. 5. Mineração de processos educacionais. 6. Reconhecimento de planos. I. Furtado, Antonio Luz. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgements

I thank my advisor prof. Dr. Antonio Luz Furtado for the guidance, inspiration and support in the development of this thesis. I also thank him for his patience when life got in the way of my studies. I also express my greatest gratitude for the discussions, related to my doctorate or otherwise, that shaped my interests these last few years.

The professors at the Departamento de Informática at PUC-Rio, especially Prof. Marco Antonio Casanova, Prof. Simone Diniz Junqueira Barbosa, Prof. Hélio Côrtes Vieira Lopes, Prof. Ruy Luiz Milidiu and Prof. Bruno Feijó for their teachings, guidance and collaboration in research. Also, thanks to all the staff at PUC-Rio for their invaluable support, especially Regina Zanon and Cosme Pereira Leal. I also thank all my past teachers, too many to name, and especially my advisors Prof. Cesar Pozzer and Prof. Angelo Ciarlini.

I thank Dell EMC, with special gratitude for my managers Ana Oliveira and Angelo Ciarlini, for the encouragement, incentive and accommodations that allowed me to pursue a Doctorate. I'm also grateful for all the wonderful experiences with my fellow researchers at Dell EMC, and also at PUC-Rio, UFRJ, LNCC, UFF, and UERJ for the various collaborations in research projects and papers in the last few years. You are too many to mention by name, but I appreciate all the help, as well as the challenges and the friendship that we have shared.

Thanks to all the researchers upon whose work this thesis builds and relies, to my friends here, home and abroad, and to my family.

Thanks to CAPES and to PUC-Rio for the granted support.

Finally, thanks to Luiza for everything.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Gottin, Vinícius Michel; Furtado, Antonio Luz (Advisor). **Discovery, Conformance and Enhancement of Educational Processes via Typical Plans.** Rio de Janeiro, 2019, 209p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In this thesis we propose the application of an automated planning paradigm based on a conceptual modeling discipline for the Process Mining tasks. We posit that the presented approach enables the process discovery, conformance checking and model enhancement tasks for educational domains, comprising characteristics of unstructured processes – with intertask dependencies, multiple dependencies, concurrent events, failing activities, repeated activities, partial traces and knock-out structures. We relate the concepts in both areas of research, and demonstrate the approach applied to an academic domain example, implementing the algorithms as part of a Library for Typical Plans for Process Mining that leverages the extensive prior art in the literature.

Keywords

Process Mining; Automated Planning; Conceptual Modeling; Educational Process Mining; Plan-Recognition.

Resumo

Gottin, Vinícius Michel; Furtado, Antonio Luz. **Descoberta, Conformidade e Aprimoramento de Processos Educacionais via Planos Típicos.** Rio de Janeiro, 2019, 209p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nesta tese propomos a aplicação de um paradigma de planejamento baseado em uma disciplina de modelagem conceitual para as tarefas de Mineração de Processos. Postulamos que a abordagem apresentada habilita as tarefas de descoberta de processos, checagem de conformidade e melhoria de modelos em domínios educacionais, que tem características de processos não-estruturados – dependências entre tarefas, múltiplas dependências, eventos concorrentes, atividades que falham, atividades repetidas, traços parciais e estruturas de nocaute. Relacionamos os conceitos em ambas as áreas de pesquisa e demonstramos a abordagem aplicada a um exemplo em um domínio acadêmico, implementando os algoritmos como parte de uma Biblioteca de Planos Típicos para Mineração de Processos que constrói sobre a extensa literatura prévia.

Palavras-chave

Mineração de Processos; Planejamento Automatizado; Modelagem Conceitual; Mineração de Processos Educacionais; Reconhecimento de Planos.

Table of Contents

List of Figures

1 Introduction

1.1 Context

Process mining is a research discipline concerned with reasoning about processes in order “to provide fact-based insights and to support process improvement” (VAN DER AALST, 2011). In a general sense, this is achieved by the extraction of knowledge from *event logs* and relying on executable process models.

The field of Process Mining defines three main tasks: *discovery* of processes, the *conformance checking* of process models to reality, and *enhancement* of process models (VAN DER AALST, 2011). All of these tasks relate to the goal of understanding the domain in order to make informed decisions, and to the potential improvement of the process itself (VAN DER AALST, DE BEER e VAN DONGEN, 2005; VAN DER AALST, ADRIANSYAH , *et al.*, 2011).

Typical approaches of process mining applications rely on graphical models, such as Business Model Process Networks (BPMN) or Petri Nets. However, certain processes possess characteristics not easily captured by such models. *Unstructured processes* yield very large graphs in these representations, even with low-frequency behavior filtered out (VAN DER AALST, 2011). The problem is aggravated as processes typically tend to be less structured than stakeholders expect (BOSE, VERBEEK e VAN DER AALST, 2011). Furthermore, these graphical models are typically ‘*flat’* representations (BOSE, VERBEEK e VAN DER AALST, 2011; GOEDERTIER, 2008) which pose inherent limitations to the *flexibility* (HEINL, HORN, *et al.*, 1999; SCHONENBERG, MANS, *et al.*, 2007) in the representation of complex behaviors. Pesic (PESIC, 2008) describes this as a primary source of incomprehensibility of models.

One aspect related to the flexibility of models and that causes difficulties for model comprehension is the representation of *intertask dependencies*: “constraints on the occurrence and temporal order of events” (ATTIE, SINGH, *et al.*, 1993).

Intertask dependencies are common in real-world processes, in which the order of execution of activities can vary significantly between cases. The process discovery task in a domain with intertask dependencies must be able to recognize partial-orderings as well as pre-requisites between activities. Furthermore, we consider another aspect of intertask dependencies – that operations can have several alternative sets of dependencies. We refer to this as the *multiple dependency* aspect.

Another class of models figures in the literature. Van der Aalst (VAN DER AALST, 2011) lists as "partial approaches" for process discovery those that rely on the *identification of rules or frequent patterns*, among which figure the *declarative* approaches for process modeling (SOWA, 1999; PESIC, 2008; PICHLER e ET AL., 2011; GOEDERTIER, 2008). The declarative approach can represent characteristics that are difficult to capture in graphical models, or that cause those graphical models to become unwieldy, such as *concurrent events*.

The declarative approach is also fitting for the representation of *failing* and *repeating* activities: in a process in which the cases may fail the execution of activities, and try them again a number of times, the results of the activities must be accounted for in the discovery, conformance checking and enhancement tasks. In the literature, the concept of *failing activities* relates to *knock-out* structures – processes in which cases can terminate in failure upon an activity-failure event (VAN DER AALST, 2001), or the absence of a success-termination event. We consider domains with *knock-out* structures based on *repeated* activity-failures.

The representation of *failing activities* and *knock-out* structures suggests a set of analyses related to the *success* of the case, particularly the mining of the *de facto* models from event logs and the success ratio obtained by the cases that follow each of those models, and the comparison of those models to the *normative pattern*. It also lends itself to the *exploration, prediction* and *recommendation* operational support activities, which relates to *online process mining*. In *online process mining* we consider not only historic information of *completed cases*, but also the "pre mortem" events, the *partial traces*, of ongoing cases (VAN DER AALST, 2011, p. 243).

1.2 Scope and Hypothesis

Our main hypothesis is that we are able to perform the typical Process Mining tasks in a domain with challenging characteristics via automated planning and a conceptual modeling paradigm.

1.3 Objectives

In this work we aim to explore a declarative approach for Process Mining, and to leverage extensive prior research on conceptual modeling (FURTADO, CASANOVA e BARBOSA, 2014; CASANOVA, BARBOSA, *et al.*, 2012) and a planning paradigm for performing the Process Mining tasks.

We particularly posit that plan-verification and plan-recognition techniques applied over a domain expressed in a conceptual model allow or support the process discovery, the conformance checking and the model enhancement.

We further posit that models defined in this conceptual model are flexible, executable, and do not suffer from the typical representational limitations imposed by representational bias (VAN DER AALST, 2011, p. 159-160). The goal, as typical in process mining approaches, is to obtain a model with an adequate representative power for the kinds of analyses desired but striving for simplicity – keeping the model as simple as possible – as well as for the other quality aspects of process models.

Hence, we don't aim to explore all of the representative power of the conceptual model or all of the features of automated planning approaches. Rather, we rely on the features and techniques that support our goal of performing the Process Mining tasks.

In regard to characteristics of the domain, we hypothesize that the application of automated planning techniques adequately enables the *online* process mining tasks for domains that comprise unstructured processes with *intertask dependencies, multiple dependencies, concurrent events, failing activities, repeated activities* and *knock-out structures*.

We instantiate and exemplify the approach in an academic domain, in which certain disciplines require the completion of others; students may fail at disciplines but take them again – possibly succeeding – in a future term; and can drop out from the academic program after certain events, including repeated failings of disciplines. We consider the students still enrolled in the program, as well as students that have graduated and dropped out. Details on the domain are given in the Appendix.

1.4 Contributions

In conciliating the research on conceptual modeling and automated planning with process mining, our work primarily extends the approach in (FURTADO e CIARLINI, 2001) for the recognition of typical plans. We enable the mining of typical plans to additionally mine constraints over attributes of events and to collect the *cases* of plans in the domain (ZAKI, LESH e OGIHARA, 2000). The cases are leveraged to compute interest metrics, which guide the model enhancement task. The model enhancement task is also guided by the discrepancies evidenced by the conformance checking, which relies on a plan-verification method.

The main contribution of this thesis is the enablement of the process mining tasks in domains with unstructured processes and challenging characteristics via automated planning and a conceptual modeling paradigm.

As specific contributions, we highlight:

- A comprehensive overview of the literature and a discussion of the links between process mining and automated planning that motivate this research, as well as other related fundamental topics;
- Defining methods for the process discovery of a three-schemata conceptual model;
- Proposing a method and developing algorithms for the plan mining approach applied to the process discovery task;
- Proposing a method and developing algorithms for the plan-verification approach applied to the conformance checking task;
- Defining and implementing algorithms for the computation of support and other interest metrics for the analysis of typical plans;
- Proposing and exploring approaches for the Model Enhancement task in the context of automated planning and a conceptual modeling paradigm.

1.5 Thesis Structure

In Chapter 2 we discuss the related work. In Section 2.1 we contextualize the field of Process Mining, discussing the seminal literature and positioning our work in relation to works that deal with similar challenges and objectives. In Section 2.2 we relate the vast body of work on the conceptual modeling of information systems. We discuss these works and the general automated planning approach in Section 2.3. In Section 2.4 we discuss works in abductive reasoning, a concept that fundaments the approaches and techniques for both automated planning and process mining, particularly regarding the process discovery task.

In Chapter 3 we discuss and present our approach for the Process Mining tasks in educational processes domains. In Section 3.1 we discuss the scope of our approach, defining the characteristics of unstructured processes that we aim to capture and manage in a Process Mining fashion. In that discussion, we identify links between process mining and automated planning that partly motivate our work. In Section 3.2 we discuss the event logs in educational domains, particularly in our example domain, and we also present the formalism that fundaments Process Mining approaches, upon which we rely to discuss concepts in the following sections of the chapter.

In Sections 3.3, 3.3.5, 3.4 and 3.5 we present our approach for the Process Mining tasks via automated planning over a conceptual model. The concepts discussed fundament the implementation of the Library of Typical Plans for Process Mining, described in Chapter 4, which we use to exemplify the applicability of the process mining via planning in an academic program domain. In Section 3.3 we discuss the definition of the conceptual model, in a three-schemata specification, and its relation to the Process Discovery task. We also provide a discussion on the domain-dependent definitions for this task. In Section 3.3.5 we discuss the mining of typical plans as part of the Process Discovery task. Section 3.4 contains the definitions of methods for the Conformance Checking task of Process Mining via plan-verification and the computation of model fitness metrics. Section 3.5 deals with the Model Enhancement task. We discuss several possible approaches, defining exemplary approaches with examples in the same academic domain.

In Chapter 4 we discuss the implementation of the Library for Typical Plans for Process mining, in which the methods discussed in the previous sections are defined as algorithms and examples in a smaller domain are given.

In Chapter 5 we present our conclusions. In Section 5.1 we perform an evaluation of our goals and hypothesis, recapitulating the discussion of prior sections. In Section 5.2 we evaluate or results against our proposed objectives and our main hypothesis. Finally, in Section 5.3 we lay out the possibilities for future developments in the topics covered by this thesis.

2 Related Work

In this chapter we investigate the literature related to the concepts in this thesis. In this thesis we discuss how we enable and support *Process Mining* via automated planning techniques, in particular regarding plan-verification and plan-recognition methods. Our motivation stems from the observation that these two areas share goals and techniques. We start by contextualizing our discussion, with respect to the kind of domain for our intended applications. We also relate works in the literature that allude to possible links between the Process Mining *tasks* and aspects of our approach for unstructured processes domains.

First, we provide the background on process mining, model representations and declarative approaches. Next, we describe the conceptual modeling discipline and the planning paradigm that supports our approach of mining of typical plans. Finally, we briefly discuss the literature respective to abductive reasoning approaches, which relates to both fields.

2.1 Process Mining

Process mining originates from both process model-driven approaches (Business Process Management – BPM, and Business Intelligence – BI), as well as from traditional data mining approaches. Seminal related work in the workflow domain was done by Agrawal (AGRAWAL, GUNOPOULOS e LEYMANN, 1998). A thorough discussion on the main concepts and applications of process mining developed since is given by (VAN DER AALST, 2011) and (VAN DER AALST, ADRIANSYAH , *et al.*, 2011).

Process mining differs from both BPM and BI in objective – it aims to provide understanding of the actual process, and not of idealized versions – and in method – it relies on process-centric techniques. One of the main inspirations for the field of process mining is that the advent of computers not only contributed to changes in organizations and processes (making them more complex), but also drastically increased the available event data (VAN DER AALST, 2011). In (VAN DER AALST e WEIJTERS, 2004) research agendas for the area are given. The same works provide an in-depth description of the three main *process mining tasks*: the *discovery*, the *conformance* and the *enhancement* of process models.

All the Process Mining tasks relate to *process models*, widely used in every sort of organization for their assistance on documenting, understanding and managing complex real-world systems. Van der Aalst describes several alternative process model representations and relates them to *event logs*, establishing a terminology that we adopt throughout this work (VAN DER AALST, 2011, p. 97-103). We highlight the following:

- A process is a collection of *activities* in terms of which the lifecycle of a single process instance is described;
- A process instance is also referred to as a *case*;
- An *event log* is assumed to contain *event* data related to a single process;
- Each *event* in the log refers to a single case;
- Each event can have many attributes, typically related to time, costs and resources.

Thus, for the purposes of the process mining tasks van der Aalst explicitly defines the following assumptions:

> "* A *process* consists of *cases.*
> * A case consists of *events* such that each event relates to precisely one case,
> * Events within a case are *ordered*,
> * Events can have *attributes*. (…)"
>
> (VAN DER AALST, 2011, p. 99)

A case is thus a process instance, and each event corresponds to the execution of one *activity* in the process regarding one such process instance.

One particularly useful aspect of process models is that they are *executable*. This typically means that they are implemented as computational simulation models that can be *run* to generate the modeled behavior. A relevant definition of computational models and their relation to simulation is given by (SOWA, 1999): every computational model is a *surrogate* for some real or hypothetical system; and, in that model, the significant entities are each assigned to variables as *surrogates* for that entity in the system. The values of those variables are *procedurally* or *declaratively* transformed to *simulate* the behavior of the system.

Given the relation between *process models* and *event logs*, the first task of Process Mining is the *process discovery* task. It consists of building a *process model* from an *event log* such that the model is representative of the behavior of the process that generated the log.

The task of *conformance checking “*relates events in a log to the activities observed in a process model” and compares them, with the goal of finding “commonalities and discrepancies between the modeled behavior and the observed behavior” (VAN DER AALST, 2011). This task highlights the need for *executable* models, so that activities representative of the processes can be observed on demand.

The task of *enhancement* extends or improves the process model using additional information recorded in event logs (VAN DER AALST, 2011). The enhancement of the model may be, for example, the *repair* of the model to correctly reflect the executions of the process as given by the log, as opposed to what a domain specialist expects. The enhancement may also comprise extending the model with additional perspectives, leveraging additional information from the process logs other than the control flow of the process.

Typically, Process Mining techniques are devised with *graphical representations* such as Business Model Process Networks (BPMN) and Petri Nets in mind (VAN DER AALST, 2011; VAN DONGEN, DE MEDEIROS, *et al.*, 2005; VERBEEK, BUIJS, *et al.*, 2010). These models are *executable*, capable of representing complex behavior (such as concurrency, for example), and are domain-independent. These models also lend themselves to algorithmic verification of several interesting properties, such as safeness, boundedness and the inexistence of ‘deadlocks’ (VAN DER AALST, 2011, p. 37-38). There are techniques for building such models that capture common execution patterns (BOSE, VERBEEK e VAN DER AALST, 2011), as well data-oriented models, and models of agents and theirs relations (like social networks or organization models (PESIC, 2008, p. 200).

A thorough discussion on various notations for process models is given by (VAN DER AALST, 2011, p. 31-57). A discussion on the quality of such models is also given (p. 128). The intrinsic tradeoff between the *fitness*, *precision*, *generalization* and *simplicity* of process models is discussed, including a discussion of behavioral equivalence (p. 141-144), and useful metrics for conformance checking. One additional important aspect of such models is the *flexibility* (HEINL, HORN, *et al.*, 1999; SCHONENBERG, MANS, *et al.*, 2007) of the representation of certain behaviors.

Since all techniques are based in some level on the event logs, the format and quality of the data contained in it are of the utmost importance. There are myriad formats for event logs – standard formats such as MXML and XES (VERBEEK, BUIJS e VAN DONGEN, 2010) have been proposed. In order to be generalizable, the techniques are often defined in relation to a more general logic formalism rather than a concrete format. We do the same under the formalism proposed by (VAN DER AALST, 2011), as discussed in Section 3.2.

Regarding the quality aspect, the completeness of the available data is important. The minimal information required in the log for Process Mining is that the (ordered) events include the *activity*. However, more information may be required for more complex representations (that is the case of our approach – also discussed in Section 3.2). The extraction of the log may typically require preprocessing steps. The literature relates several practical challenges for systematically extracting event logs from real-world data (VAN DER AALST, 2011, p. 112-114).

2.1.1 Declarative process mining

Declarative process models have been also proposed for process mining (PESIC, 2008; CHESANI, LAMMA, *et al.*, 2009). One of the main reasons for the utilization of declarative process models is to cope with the limited flexibility of more imperative approaches (PICHLER e ET AL., 2011). For example, (CHESANI, LAMMA, *et al.*, 2009) present a method for the discovery of declarative constraints.

Many works on declarative process mining figure in the literature. Sowa contrasts a procedural and a declarative approach to the *executions* of executable models: the “procedural approach uses programs or rules that operate on the variables of model M”, while the “declarative approach is based on constraints or *axioms* that define the *preconditions, postconditions*, and *transformations* for each event that may occur in the model” (SOWA, 1999, p. 135).

Goerdertier (2008) describes a method for the declarative process discovery in which negative events are generated for inductive learning, as in (FERREIRA e FERREIRA, 2006). Pèsic (2008) describes the Declare system. Declare comprises a workflow management system, using a graphical notation for models, and a declarative process modeling language for defining linear temporal logic (LTL) constraints. The system allows for mining and verification of constraints, and draws from the SCIFF framework (ALBERTI, CHESANI, *et al.*, 2006) for abductive logic-programming (CHRISTIANSEN, 2009; KAKAS, KOWALSKI e TONI, 1998). We relate the literature in abductive reasoning and the abductive logic programming paradigm in Section 2.4.

Different taxonomies for the characterization of flexibility in process models exist in the literature (HEINL, HORN, *et al.*, 1999; SCHONENBERG, MANS, *et al.*, 2007). In the taxonomy proposed by Schonenberg two important aspects are *flexibility by design* and *flexibility by under specification*. *Flexibility by design* relates to the “ability to specify execution alternatives in the process model” (PESIC, 2008, p. 22). This characteristic is present in the rule-based and constraint-based process modeling languages (DOURISH, HOLMES, *et al.*, 1996; GLANCE, PAGANI e PARESCHI, 1996; WAINER e DE LIMA BEZERRA, 2003). *Flexibility by under specification* relates to the model’s ability to leave parts of execution alternatives unspecified, to be defined later (via *late modeling*) during execution. Chesani et al. additionally present a constraint-based declarative process mining approach. Their approach relies on Inductive Logic Programming (CHESANI, LAMMA, *et al.*, 2009) to mine models expressed in the ConDec notation proposed by Pèsic (PESIC, 2008).

2.1.2 Frequent itemsets and sequence mining

Process mining also closely relates to the field of *frequent sequence mining* (VAN DER AALST, 2011, p. 77-81), particularly regarding the approaches for *episode mining* (MANNILA, TOIVONEN e VERKAMO, 1997). These approaches stem from the frequent itemset literature and shares algorithmic implementations.

The approach for defining frequent itemsets (AGRAWAL e SRIKANT, 1994) is typically described in terms of a database of transactions from a multitude of consumers, each transaction in turn comprising one or more items bought in that transaction. This constitutes the so-called ‘market basket analysis’ problem, in which we aim at the discovery of items that are frequently bought together (ZAKI e MEIRA JR., 2014). Under this interpretation, the *support* of an itemset is defined as the proportion of all transactions in the database containing that itemset. The itemset is considered *frequent* if its support is higher than a predefined minimum support threshold value.

The identification of all frequent itemsets in a database enables the generation of *association rules* among those itemsets. An association rule relates two itemsets, A and B, expressing that whenever a transaction contains the items in A, it most likely contains the items in B. More formally, An association rule $A\xrightarrow{s,c}B$ is a binary relationship between two disjoint itemsets, A and B, determining a support *s* and a confidence *c* relating to the transactions in which A and B appear (HORNIK, GRÜN e HAHSLER, 2005). The support *s* of the rule comprises the count of transactions in which A and B co-occur (that is, the support of itemset A∪B). The confidence *c*, on the other hand, determines the conditional probability that a transaction in the database contains A given that it contains B (that is, the *support* of itemset A∪B divided by the support of itemset A). Like in the case of itemsets, the association rule is *frequent* if its support is higher than a predefined minimum support threshold value. Conversely, the rule is *strong* if its confidence is higher than predefined minimum confidence threshold value. Typically, we’re interested in rules that are both frequent and strong, but this depends on the domain. More thorough formal definitions are given by (ZAKI e MEIRA JR., 2014, p. 217-220).

A plethora of techniques and algorithms have been proposed for the task of association rule mining (and, by extension, frequent itemset mining). Most prominently in the literature, the Apriori algorithm (AGRAWAL e SRIKANT, 1994) is based on exploring the downward closure property – which guarantees that frequent itemset has an infrequent subset – for generating increasing candidate itemsets. Apriori spawned a family of algorithms, including variations on how itemsets and transactions are represented (AprioriTID and AprioriHybrid (AGRAWAL e SRIKANT, 1994), SETM (HOUSTMA e SWAMI, 1995)); on intertwining the candidate generation and support counting steps (DIC, (BRIN, MOTWANI, *et al.*, 1997)); and on how the database is loaded in memory, concerning size limitations (Partition, (SAVASERE, OMIECINSKI e NAVATHE, 1995)). Besides Apriori, depth-first search approaches have been developed as in the FP-growth algorithm (HAN, PEI e YIN, 2000) , which does not require an explicit candidate generation step, and the Eclat approach (ZAKI, PARTHASARATHY, *et al.*, 1997), which combines the depth-first search approach with the tidlist representation from AprioriTID.

2.1.3 Educational process mining

The domain use case for the empirical demonstration and evaluation of our approach is an academic program. Thus, we relate the prior art in educational process mining, a subset of the process mining literature that deals with this specific domain. We refer to our previous work regarding typical conformance checking over an academic curriculum as a process (GOTTIN, JIMÉNEZ, *et al.*, 2017). Further work on this area, relating to the same use cases, figures in (JIMÉNEZ, 2017).

The representation of curricula and academic processes as graphical process models (Petri Nets, specifically) is explored by (JUHÁSOVA, KAZLOV, *et al.*, 2016). They propose to deal with the complexity that originates from “curriculum rules, mandatory courses, prerequisites and causal dependencies between courses, minimal number of credits needed to earn a degree” by creating “subnets” (idem). However, the resulting models, as exemplified in the paper, can become very complex, comprising dozens of places at the top-level even when ordering constraints are omitted for the sake of simplicity. The example Petri Net by Juhásova et al. does not cover failures and re-attempts at disciplines, for example.

Constraint-based and integer linear programming (ILP) approaches are considered for the representation of curricula by (HNICH, KIZILTAN e WALSH, 2002) in the context of the “balanced academic curriculum problem”. The problem is defined as “to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced i.e. as similar as possible" while still "following administrative rules and academic regulations" (HNICH, KIZILTAN e WALSH, 2002, p. 2)). Constraint-based, ILP and hybrid solutions are considered for this problem, regarding the ease of representation, the optimality and the provability of the solution.

A thorough recent survey on Educational Process Mining is given by Bogarín et al. (2018). Many related works are concerned with activities in a granular scope, e.g. regarding Learning Management Systems (LMS): "Log files generated by these systems provide an insight into how people follow the course, when they watch videos or lectures, and when they hand in tasks, among others." (BOGARÍN, CEREZO e ROMERO, 2018, p. 9)). Chesani et al. (2009) posit the discovery of declarative (constraint-based) models in an educational domain as possible future work. Bellodi et al. tackle this (BELLODI e LAMMA, 2010) in the same context, considering dropped-out students as negative examples for the discovery of integrity constraints. These works also introduce the terminology of a student's *career* as the trace of the student case in the academic process.

Also related to our approach are the works on *Curriculum Mining*. The goal of curriculum mining as defined by Bogarín et al. is to "gain a better understanding of the underlying educational processes" (2018, p. 12). This relates directly to our goal of discovering the typical plans adopted by students, and somewhat relates to the comparison of these plans to the recommended curriculum. A first paper concerned with curriculum mining and conformance testing is (TRCKA e PECHENIZKIY, 2009). They rely on the representation of Colored Petri Nets (ZIMMERMANN, 2008), defining 'basic curricular constraint patterns' based on pre-authored pattern templates. They state that they were able: to check for known constraints; to analyze "how much time and effort a particular activity takes"; and to "predict student dropout". However, they do not state how they obtain the curriculum templates other than relying on domain knowledge. Meanwhile, they list as 'ongoing work':

> "(…) finding most common types of behavior and clustering them, finding emerging patterns that capture significant differences in the behavior of students who graduated vs. those students who did not or significant changes in behavior of students from one generation

> and the other; finding frequent patterns that describe a bottleneck in the curriculum, i.e. patterns explaining for which students it is the bottleneck and why."
>
> (PECHENIZKIY, TRCKA, *et al.*, 2012).

Finally, Wang and Zaïane also aim at discovering a curriculum process, and similarly to our approach, to "compare cohorts of students, successful and less successful" and present an opportunity to "adjust the requirements for the curriculum by applying enhancement of process mining." (WANG e ZAÏANE, 2015). However, they also rely on a Colored Petri Net. Interestingly, Wang and Zaïane evidence the problems with intertask dependencies in the representation of curricula as processes: "other requirements include the first and the last course a student must take, mandatory courses, and non-coexisting courses, i.e. if the student takes one course in the group then they cannot take any other course belonging to the same group" (idem). Due to the inherent complexity of the representation, they generate separate models for each class of interest, covering only the 'most frequent activity paths', as "the model map would be too dense and cluttered to recognize patterns if we present all of them" (idem, p 518).

Van der Aalst (VAN DER AALST, 2011, p. 290) briefly discusses opportunities of process mining applications in the education sector, highlighting the difficulties that rise from students having very different study patterns. The aspect of failing activities is also considered: "the design of a curriculum should not only focus on the "ideal student" (that passes all courses the first time), but also anticipate problems encountered by other students" (idem).

2.2 Conceptual modeling and the Three-schemata specification

We refer to (FURTADO, CASANOVA e BARBOSA, 2014; CASANOVA, BARBOSA, *et al.*, 2012) for a survey of this and multiple related research topics:

> "Our understanding of information systems comprises facts, events and agents. Everywhere the Entity-Relationship model is used. The existing entity instances and their properties, i.e., their attributes and the relationships among them, are the *facts* that characterize a state of the world. States are changed by the occurrence of *events* caused by operations defined by pre-conditions and post-conditions that are in

turn expressed in terms of such facts. The event-producing operations are performed by certain *agents*, in an attempt to satisfy their goals, once again expressed by facts. Accordingly, our specifications are divided into three schemas to introduce, respectively, the classes of facts (static schema), events (dynamic schema) and agents (behavioural schema)."

(FURTADO, CASANOVA e BARBOSA, 2014)

Here, we focus on describing the three-schemata specification – encompassing the *facts*, the *events* and the *agents* in the system - and its main concepts that fundament the application of *automated planning* techniques. A formal definition of the three-schemata representation is given in (CIARLINI, CASANOVA, *et al.*, 2010).

The *static* aspect of a system can be typically captured by a factual database. Denoting the facts that characterize the entities and their properties as predicates, a *state* is the set of all predicate instances holding at a given instant of time. Thus, a *state* is a description of the world underlying the database at that instant. The *dynamic* aspect of the system characterizes the actions performed in the world. These actions are denoted as *operations*, defined in regard to their pre- and post-conditions, as in the STRIPS formalism (FIKES e NILSSON, 1971). In that formalism, pre-conditions indicate the facts that must hold in the state at which the operation is executed - the prerequisites for the action that operation represents. Conversely, the post-conditions indicate the facts (affirmed or negated) that hold in the state following the operation – that is, the effects of the operation. Finally, the *behavioural* aspect captures the agents and their motivations. These motivations are modeled through *goal-inference rules* associated with each agent or class of agents, modeling their predicted behavior under certain circumstances. Each such rule indicates a goal (expressed as facts that should hold or cease to hold in a goal state) that said agent is motivated to reach when a certain situation (also expressed as facts holding or not holding, at a current state) is encountered.

A concept related to goal-inference rules is that of ECA (Event Condition Action) rules that take the form "ON Event IF Condition DO Action" (PAPAMARKOS, POULOVASSILIS e WOOD, 2007). However, goal-inference rules differ in that they model the intention of the agent or class of agents, instead of 'trigger rules', by representing the *goal state* rather than explicitly the actions to be performed. Furthermore, in order to achieve the goal state, a goal-inference rule potentially 'fires' *sequences* of operations, instead of a single atomic action.

Particularly regarding narrative contexts, many works have been published on the interpretation and generation of plots, both for entertainment and serious games domains (CIARLINI, CASANOVA, *et al.*, 2010; CIARLINI, BARBOSA, *et al.*, 2007; GOTTIN, DE LIMA e FURTADO, 2015). It should be noted that the serious games aspects of that work were initial steps towards process mining applications.

Other works in conceptual modeling relate to the semiotic approach described in (FURTADO, CASANOVA e BARBOSA, 2014). Others still relate to database prototype and (re-)design, additionally regarding temporal databases and planning (FURTADO e CASANOVA, 1990) and their relation to narratives (FURTADO, 1999). A method to construct running conceptual level specifications, all the way to an implementation in a database management system is described in (GOTTIN, DE LIMA e FURTADO, 2015).

The schema definitions, when represented as Prolog clauses, configure an *executable* conceptual specification. This specification lends itself to the applications of algorithms for planning and for the simulated execution of plots: the representation of the domain in three-schemata is "integrated through the application of a plan-recognition / plan-generation paradigm." (CASANOVA, BARBOSA, *et al.*, 2012, p. 24). We will focus our discussion throughout this thesis on the plan-recognition that we leverage in this work (CIARLINI e FURTADO, 2002). The executable aspect of the specification allows its use as a simulation model (CIARLINI, 1999) and for the replay of plans.

The plan mining approach leveraging the goal-inference rules is based on plan-recognition and as such characterizes an *abductive* mode of reasoning. We discuss related works in Section 2.4.

2.3 Automated planning

Research and applications in automated planning (GHALLAB, NAU e TRAVERSO, 2004) are numerous. Here, we focus on the relevant that is closely related to our work.

We represent operations in the STRIPS representation. That representation, along with other so-called *classical representations*, inspired and guided the development of PDDL (*Planning Domain Definition Language*) as a common formalism for expressing "the 'physics' of a domain, that is, what predicates there are, what actions are possible, what the structure of the compound action is, and what the effects of the actions are" (MCDERMOTT, GHALLAB, *et al.*, 1998). Notice, however, that in the literature related to PDDL the term *process* is used with a different meaning than that of Process Mining – rather than sequences of activities in general, a process is 'autonomous', i.e. activities that go on independent of the executor of plans' actions (MCDERMOTT, 2003).

The seminal work of Kautz in plan recognition (1991), as well as Yang et al.'s Abtweak (YANG, TENENBERG e WOODS, 1996), are leveraged by Ciarlini et al. for modeling interactive storytelling genres as application domains (CIARLINI, CASANOVA, *et al.*, 2010). Developments in this area regarding plan recognition are further described in (FURTADO e CIARLINI, 2001), (CIARLINI, 1999) (including goal recognition techniques) and (FURTADO e CIARLINI, 2000). These works are also related and based partly on the conceptual modeling definitions by (FURTADO, CASANOVA e BARBOSA, 2014) and (CIARLINI, CASANOVA, *et al.*, 2010) described above.

The plan-generation process relies on the inference of goals from agents, as given by the behavioral schema. All such rules are evaluated at a predetermined initial state, in order to infer the agent's goals in the situation that state represents. In (CIARLINI, 1999) a plan-generation algorithm, based on the same conceptual model that we use in this thesis, is employed to generate possible sequences of operations that lead to a state in which the goal is attained. The plan-generation is part of the IPG system and is based on Abtweak (YANG, TENENBERG e WOODS, 1996).

The plan-recognition process, on the other hand, concerns the identification of possible plans agents are trying to perform (KAUTZ, 1991; CIARLINI, 1999). That is, from the observation of a non-necessarily consecutive series sequence of events, plan-recognition algorithms can identify the plans being executed by agents in the system. Relying on *goal-inference rules*, as described above, it is possible to further hypothesize and reason about the possible goals being pursued by the agents

(CIARLINI, 1999). At each intermediate state (from the initial state to the goal state, after each operation) further goal-inference rules can be evaluated. This can be used to represent interactions (such as collaboration or competition) between agents in the system. As usual in planning applications, constraints set by the initial state and the applied operations are enforced.

The literature in automated planning raises several topics that relate to our approach. One approach that relates to the mining of a planning domain from event logs, similar to a process discovery task, is the *automatic domain knowledge acquisition* (CRESSWELL, MCCLUSKEY e WEST, 2013)*:* "The prevalent idea in Automated Planning research and development is that there is a logical separation of planning engine and domain model representing the application and problem at hand. However, these domain models are invariably hand crafted". In that approach a planner aims to learn, from an incomplete definition, a full definition of the planning domain by experimentation (GHALLAB, NAU e TRAVERSO, 2004). The relationship between graphical models (such as petri nets or business process model networks) and classical planning domains is explored in (HICKMOTT, 2006; STRYCZEK, 2008; GONÇALVES e BITTENCOURT, 2005). There is also prior work on the relationship between graphical models and other planning domain formulations, such as Hierarchical Task Networks (GONZÁLEZ-FERRER, FERNÁNDEZ-OLIVARES e CASTILLO, 2013). Finally, the work on derivational analogy (VELOSO e CARBONELL, 1993) relates to the building of a library of typical plans.

As previously mentioned, our work is based on prior research in conceptual modeling which has been applied to several other contexts. Several planning techniques have been specialized for adoption in these contexts (CIARLINI, 1999; DA SILVA, CIARLINI e SIQUEIRA, 2010; ABELHA, GOTTIN, *et al.*, 2013). Some of these works concern the presentation of plots in graphical, textual and multimedia format (RODRIGUES, POZZER, *et al.*, 2015; DE LIMA, 2014).

There are some works relating aspects of the automated planning literature and the field of process mining. In (CRESSWELL, MCCLUSKEY e WEST, 2013) a method for the *discovery* of a planning domain from examples is given. A similar approach is taken by de Leoni and Marrella (2017) who define a method to build a planning domain and problem instance via the encoding of an alignment problem in Petri Nets (VAN DER AALST, 2011). Heinrich and Schon (2015) tackle process environmental factors in planning for BPMN. Awad et al. (2009) employ automated planning to address the semi-automatic resolution of violations in execution ordering compliance rules. Our work differs from these in that we leverage a known schema of the domain, known a priori, and also perform the discovery of typical plans of classes of cases for conformance testing and model enhancement. Our approach is thus able to perform the three main tasks in Process Mining, considering domains with generalized *knock-out* structure as well as other interesting and challenging characteristics.

2.4 Abductive Reasoning

The term *abduction* originates from the works of Peirce, circa 1900, who characterized it as one of three forms of logical inference, along with *induction* and *deduction* (CONSOLE, DUPRÉ e TORASSO, 1991; MCILRAITH, 1998; ESHGHI e KOWALSKI, 1988). Contrary to the deductive modality of reasoning, both abduction and induction are not completely certain – induction being the production of a rule from many observations, and abduction being the production of an explanation given rules and observations. The abductive modality of reasoning is characterized, in the field of Artificial Intelligence (AI), as the process of forming explanatory hypothesis. As such, it is intrinsically related to the methods employed in the plan-recognition approaches. It is also related to the task of *process discovery*, since it derives a model from observations – in the form of an event log.

Since the introduction of the concept by Peirce, circa 1900, as the *selection of a preferred explanation* for an observed occurrence, it was popularized within the field of AI in the 1970s, a number of research groups have dedicated efforts to the production of engines of abductive reasoning, with the goal of providing the mechanization of the *explanation* and *prediction* capabilities of the abductive modality of reasoning. Kakas et. al state that “over the last two decades it has become clear that abduction can play a central role in addressing a variety of problems in Artificial Intelligence” (KAKAS, A. C.; NUFFELEN, B. V.; DENECKER, M., 2001). A thorough account of Peirce’s thoughts on the topic of abduction is given in (FANN, 1970). Peirce’s work has influenced many fields of study, with applications in philosophy, history, anthropology and medicine, as well as in computer science, predominantly regarding artificial intelligence (LLERA, 1997; KAKAS, KOWALSKI e TONI, 1998; GABBAY e WOODS, 2005; PAAVOLA, 2004; POOLE, GOEBEL e ALELIUNAS, 1987; PENG e REGGIA, 1990) after the popularization of the term by Pople (POPLE, 1973).

McIlraith (MCILRAITH, 1998) identifies three categorizations of the topic of abduction. The first one is the *set covering* account, wherein the association of explanation primitives and manifestations (observations) is represented in a mathematical framework (REGGIA, 1983; ALLEMAND, TANNER, *et al.*, 1987). In this characterization, abduction is defined over sets of observations and hypothesis (KAKAS, KOWALSKI e TONI, 1992). This approach is used for the explanation of diagnostic problems, although with limited expressive power (MCILRAITH, 1998; REGGIA, 1983; EITER, GOTTLOB e LEONE, 1997).The second account is *probabilistic*, wherein the plausibility (likelihood) of an explanation – a means of preference between distinct explanations - is computed based on prior knowledge of the probabilities of all abducible explanations for observations (PEARL, 1998; PENG e REGGIA, 1990). Probabilistic frameworks for abductive reasoning integrate this notion of plausibility with causal networks, where the probabilistic relationships are explicitly stated. This of course configures the problem of estimating these probabilities, which can be unfeasible in many domains. Regardless, this account has considerably grown in importance with the advent of big data applications of machine learning, wherein massive causal networks with associated probabilities can be mined from real-world processes (MOONEY, 2000; VAN DER AALST, 2011). Finally, the third account of abductive reasoning according to, established in the field of artificial intelligence, is that of *theory formation*, a formalization of the concepts of a theory as premises, observations and abducibles as sentences in a language $\mathcal{L}$ (KAKAS, MICHAEL e MOURLAS, 2000; POOLE, GOEBEL e ALELIUNAS, 1987; CONSOLE, DUPRE e TORASSO, 1986). Several systems implement mechanism of abductive reasoning, typically within the framework of Logic programming systems.

3 Process Mining via Typical Plans

In this chapter we describe our proposed application of an automated planning paradigm based on a conceptual modeling discipline for the process mining tasks of Process Discovery, Conformance Checking and Model Enhancement in educational processes domains. In this chapter we define and formalize the methods, whose implementation is discussed - in the next chapter – as part of the Library of Typical Plans for Process Mining.

3.1 Unstructured processes and Automated Planning

Academic processes are *unstructured*, with challenging characteristics (see Chapter 1). We aim, leveraging our conceptual modeling discipline, to adequately capture the domain of such processes. As shown in many related works, this conceptual model can be leveraged for automated planning techniques that provide a high level of *flexibility* in the reasoning and representation of the domain. We extend the approach in (FURTADO e CIARLINI, 2001) towards the application of these techniques for Process Mining tasks.

We explore the planning paradigm motivated by a high level of *flexibility by design* it provides. In that paradigm the description of operations via their preconditions and effects naturally represents dependencies between activities without constraining a specific order – rather, it establishes *partial-order* dependencies. This is similar to the motivation behind declarative model specifications, such as (WAINER e DE LIMA BEZERRA, 2003), that represent dependencies between events via sets of conditions that must hold before and after each activity. While not related specifically to declarative process mining, the approach by (FERNANDES, CIARLINI, *et al.*, 2007) also relates to this type of flexibility, relying on incremental planning.

The concept of flexibility by under-specification also relates to the characteristic of the automated planning paradigm of allowing for the selection of the most appropriate alternative at runtime. This corresponds to the concept of an *executable model*, allowing for the *simulation* of the process, in the Process Mining literature. Many prior works have highlighted the simulation aspect supported by *plan-generation* automated planning techniques (FURTADO, CASANOVA, *et al.*, 2007; CIARLINI e FURTADO, 2002). The *plan-verification* task that we employ is also supported by the executable aspect of the planning model.

This "executable" characteristic also relates to Goedertier's definitions of *state*- versus *goal*-driven approaches:

> "Unlike procedural process modeling, declarative process modeling does not involve the pre-computation of task control flows, information flows and work allocation schemes. Whereas procedural process models inherently contain pre-computed activity dependencies, these activity dependencies remain implicit in declarative process models. An explicit enumeration of all activity dependencies is often not required – and often even difficult to obtain (HEINL, HORN, *et al.*, 1999). For model checking (verification) purposes, execution trajectories can still be obtained from implicit process models. During the execution of a declarative process model, a suitable execution scenario is constructed (either by a human or machine coordinator) that realizes the business goals of the process model. **The latter is called goal-driven execution and its automation is akin to planning in the domain of Artificial Intelligence** (GHALLAB, NAU e TRAVERSO, 2004). In contrast, the execution mechanism of procedural process modeling languages is called state-driven."
>
> (GOEDERTIER, 2008, p. 15)

The flexibility of models in the automated planning paradigm is evidenced by the fact that it naturally avoids typical representational limitations, as posited by (VAN DER AALST, 2011, p. 159-160), typically associated with the discovery, and thus the representation, of process models.

Automated planning models are able to represent concurrency via the representation of time in the definition of operations and the ability to deal with partial orders in plans. In the educational domain we use as examples, it is typical for students to perform several activities in the same academic term. The models can also deal with skipped activities, which are naturally accounted for in the plan recognition mechanisms. In the educational domain example, it is typical for students to skip optional disciplines, or to choose between alternative disciplines when both satisfy pre-requisites for advancing in the program.

Recall from Chapter 1 that we are concerned with domains with *repeated activities*. The representation of failing activities is naturally captured by the proper definition of operations' effects in planning approaches. The same, along with the proper definitions of operations' preconditions, is true for the representation of OR-splits and joins, and of non-free-choice behavior. Furthermore, planning techniques are naturally suited for the representation – and reasoning about – high level operations. Techniques for handling hierarchical task networks in automated planning, and the *complex* plans, as in (FURTADO e CIARLINI, 2001) and herein, are both examples of how these high-level tasks can be accounted for in planning applications.

Several prior works have raised issues, indicated therein as possible future works, that relate to our approach. In (CIARLINI, 1999) the discovery of typical plans as a knowledge discovery task, relying on plan-recognition, is initially posited. Later works that developed this concept (FURTADO e CIARLINI, 2001; FURTADO e CIARLINI, 2000) provide the fundamental approach over which we build our Library of Typical Plans for Process Mining, and raise the issue of determining typical plans based on frequency of occurrence. The usage of the frequency of occurrence of plans as a "confirmation criterion" is also envisioned, as discussed in Section 3.4. Another aspect envisioned in these seminal works is the automatic generation of goal-inference rules. We discuss that in Section 3.5.

In (FURTADO, CASANOVA, *et al.*, 2007) figures a thorough discussion on how *plot analysis* over database logs that register the actions of individual agents can provide "a rich source of knowledge about the agent's behavior". This highlights that the analysis of plots as sequences of events extracted from logs was already contemplated in the literature. That work also considers how the goal-inference rules in the behavioral schema provide insight into agent's behaviors: "Plots involving a given agent provide one significant indicator of the agent's behaviour, and, as such, can be used for characterization and comparison purposes". We relate that work to our developments in *process model enhancement*, particularly regarding decision mining.

Finally, we have explored the analysis of academic programs (degree curricula) through Frequent Itemset analysis and Process Mining analysis (GOTTIN, JIMÉNEZ, *et al.*, 2017). Our conclusion regarding typical Process Mining application, using the tools offered by the PROM software, mostly geared towards graphical models, was that these were not ideal. We noticed that the characteristics of partial-order between the disciplines generates too much variability in the traces of students to easily capture in graphical models. Also, the knock-out structure, with failing activities and non-successfully terminating processes, makes it hard to reason about the domain without separating success and failure cases a priori. Enabling the Process Mining tasks over such a domain is the main goals of our approach.

Our goal in this work is to support the enactment of the Process Mining tasks via the planning paradigm that has been extensively explored in the literature. To that end, we rely on plan-recognition and plan-verification. We will provide the definitions that support the enactment of the Process Discovery task in Section 3.3 and 3.3.5; of Conformance Checking in Section 3.4; and Model Enhancement in Section 3.5. In those sections, we seek a good tradeoff between representing details of the domain to and adequate level and providing a simpler, more general, definition of the approach. Hence, in several aspects, our formulation – at least initially – is simpler than the conceptual modeling discipline allows. Recall our discussion from Chapter 2 - this is in line with the *simplicity* quality criterion of process models, which states that "the discovered model should be as simple as possible" (VAN DER AALST, 2011, p. 128). Regardless, the Library of Typical Plans for Process Mining that is implemented retains the capabilities of the prior work, as discussed in Chapter 4. We provide a discussion on several domain-dependent aspects of our general formulation in Section 3.3.4. Finally, we increment the behavior captured by the model through the Model Enhancement task in Section 3.5.

3.2 Event logs and the example educational domain

Process mining is intrinsically related to *logs* and how they are defined. In order to compose an executable model with which to reason about, Process Mining tasks require information on relevant events and their order. The seminal literature (VAN DER AALST, 2011, p. 97-99) defines a series of chrematistics of the process and the required information in the log; we now relate these definitions to our educational domain example. A detailed description of our use case domain and its terminology is given in the Appendix.

The *academic program* constitutes the *process*. Each *student* is a *case* in the process, performing *disciplines* as the *activities*:

- "A *process* consists of *cases*" –
  The academic program process consists of student cases.

- Scoping: "one event in the log corresponds to one process. Only relevant events are in the data":
  Each event in our log corresponds to a single *student*.

- "All mainstream process modeling notations (…) specify a process as a collection of activities such that the lifecycle of a single instance is described":
  The academic program is a collection of disciplines required for graduation. The performance of a single student across disciplines is the lifecycle of the student.

- "A case consists of *events* such that each event relates to precisely one case" –
  This relates to the "scoping" of the log. An *event* in the academic program process describes the (attempted) completion of a *discipline* (a course) by one student. The *disciplines* correspond to the *activities* in the process.

- "Events within a case are *ordered*" –
  The disciplines performed by a student are ordered. However, typically a student typically performs several disciplines in each academic term (semesters, in our case domain).

- "*Events* can have *attributes*" –
  The attributes of the events include the timestamp (the year and academic term), the result of the activity – success or failure, the grade obtained by the student, the professor, and others.

The minimal information required in each event in the log for the Process Mining tasks are the *case id* and the *activity*. The case id links the event uniquely to a case instance. The activity is required to characterize the control flow perspective of the process. The sequence of unique events for a case comprises a *trace*.

We show a fragment of the log of our educational domain example in Table 1. This particular (anonymized) dataset originates from the database of an actual educational process, with data comprising the traces of students in the Computer Science program at PUC-RIO. Its preparation required preprocessing steps, as is typical in process mining (see Section 2.1). The preprocessing stages performed to obtain this event log, including anonymization, filtering, grouping and scoping, are given in references listed in the Appendix. Here, we focus on the relevant concepts relating to the event log definitions for our case.

Table 1 A fragment of the event log.

Event id	**Case id**	**Event Attributes**					
		timestamp	**activity**	**credits**	**class**	**lecturer**	**grade**
77273	2368	8	INF1413	4	3WA	Jessica Leon	57
77417	2370	7	INF1636	4	3WA	Crystal Landry	57
78755	2389	6	INF1406	4	3WA	Crystal Landry	100
78829	2390	4	FIS1033	4	33A	Jonathan Jacobson	17
…	…	…	…	…	…	…	…

To begin with, in the Process Mining fashion, we must uniquely identify the *cases*. We uniquely identify our students (*cases*) by the Case id – the student's unique enrollment number. In this domain, an event is the attempt by a student to complete one of the *activities* (disciplines). As typical in Process Mining fashion, the log is prepared such that each row comprises an *event* – with a unique Event id, generated in the data extraction process. Thus, each row contains the description of an event - an *activity instance* of a student (our *case*) performing a 'discipline' in the Computer Science program (the *process*) at a certain *timestamp* (the academic term). The *timestamp* of the event is the time associated with the *event* – not necessarily corresponding the timestamp of the 'recording operation' in the information system that records the event, e.g. the timestamp of a database transaction.

The event log initially considered contains the traces of 23 students that share the academic term of first enrollment. Later, in Section 3.5, we will consider an additional set of students. The Appendix also describes the terminology and concepts of the domain in more detail.

In the most general formulation of Process Mining, an Event id is not explicitly available, and the events must be uniquely identified by the event attributes. The minimum set of event attributes that uniquely identifies events defines the *event classifier*.

In simple processes where each *activity* is executed at most once, it is typical to refer directly to the *event* by the *activity* name. Consider, for example, a simple process where activities ‘a’, ‘b’ and ‘c’ are involved, and every case performs exactly two of those activities, without repetitions. For a particular case ‘X’ that performs activity ‘a’ followed by ‘b’, we can refer to the first *event* in the *trace* <a,b> of case X by the activity name unambiguously. Consider, in contrast, a similar domain where activities can be repeated. A particular case ‘Y’ that performs activity ‘a’ twice and then ‘b’ twice yields a *trace* <a,a,b,b>. Now, in order to refer to one of the events unambiguously we must use additional information other than just the activity name. In this case, we could use the ‘order’ of the events in the log – “the first execution of activity ‘a’”, for example.

In our educational domain, a student (*case*) may attempt to complete a discipline (*activity*) but fail. Then, the same student may enroll in the same discipline again in a future semester. This highlights that in our domain, we need to refer to events not only by the *activity* name, but also by the *timestamp,* the academic term – i.e. the semester – in which that *activity instance* took place. Notice also that many events of a same student (*case*) may have the same timestamps, as it is typical for a student to take many classes in the same semester - however, the student cannot attempt the *same* discipline twice at the same academic term. Hence, the discipline (*activity*) and the academic term (*timestamp*) are both required and sufficient to uniquely identify events in our domain. We recapitulate this example below, when formalizing the concept of the *event classifier,* which yields the unique names for each event in the event log.

Similarly, as for the Event id, an explicit Case id for each event is not a hard requirement for Process Mining approaches. If the cases can be uniquely identified, these unique identifiers can be “generated when extracting the data from different data sources” (VAN DER AALST, 2011, p. 105). For ease of representation, in the discussion on the formalization of our approach, we assume that identifier values are available.

Besides the *timestamp* and the *activity*, the *credits*, *class*, *lecturer* and *grade* are also *event attributes*. In the Process Mining literature these additional event attributes are typically used for the tasks of *process enhancement*.

We highlight a special kind *of event attribute* that relates to the *result* of the activities. Since we aim at representing successful and failed attempts at disciplines by students, we incorporate these attributes in the formalism of our conceptual model. This is further discussed in the formalization below, and in Sections 3.3 and subsections.

In the educational domain, the *Grade* constitutes such an attribute of special interest to our analysis. Clearly, the final grade obtained by the student determines success or failure. The event log above does not show the available *case* attributes. We assume a single case attribute $status$ that determines, for the students that have already left the academic program, whether they graduated, dropped out or are currently enrolled in the program. This is the *case termination* status. We will consider additional case termination statuses in Section 3.5.

One particularly interesting aspect of event logs is the possibility of extracting them from ordinary database management systems. Many approaches for the extraction of logs from databases for process mining figure in the literature (RODRIGUEZ, ENGEL, *et al.*, 2012; VAN DER AALST, 2015) – these require a *history* of the database: “the “regular tables" in a database only provide the current state of the information system. It may be impossible to see when a record was created or updated. Moreover, deleted records are generally invisible. Taking the viewpoint that the database reflects the current state of one or more processes, we define all changes of the database to be events.” (VAN DER AALST, 2015).

The approach on which we base our work also considers the extraction of sequences of events from logs (CIARLINI e FURTADO, 2002). Alternative methods for semi-automatic conceptual model definition, which are *not* based on event logs, can be found in the literature, involving in particular the composition of schemas from previously available schemas. One such proposal, relying on analogy and generalization, is given in (FURTADO, BREITMAN, *et al.*, 2008).

In the following, we adopt and extend the formal definitions from (VAN DER AALST, 2011).

Let

- $\mathcal{E}$ be the set of event identifiers in the domain;
- Let $\mathcal{C}$ be the set of all possible case identifiers;
- AN be the set of all attribute names.

Both events and cases are characterized by attributes.

For each event $e \in \mathcal{E}$ we define $\#_n(e)$ to be the value of attribute $n \in AN$ for event e.

Similarly, for each case $c \in \mathcal{C}$ we also define $\#_n(c)$ to be the value of attribute n for case c. Let $case\_id$ be an attribute that uniquely identifies the case.

An absent (null value) attribute is given by $\perp$.

The values for the attributes of an event typically include $\#_{activity}(e)$ and $\#_{timestamp}(e)$ attributes, but only $activity$ is mandatory. The $timestamp$ attribute can be absent in some domains since the event log is ordered. It is typically required, however, to reason about concurrent events, or events with a complex activity lifecycle.

In our case, we assume that the $timestamp$ attribute is present. It is used to disambiguate between distinct attempts at the same activity, as we discuss next.

We also assume that the $case\_id$ is present, as in typical Process Mining fashion.

Events are identified in the log by a unique identifier – a *name*. A *classifier* determines a name $\underline{e}$ for each event $e \in \mathcal{E}$. Recall that the bare minimum information for an event typically includes only the activity - this corresponds to the *default classifier* $\underline{e} = \#_{activity}(e)$, which allows to refer to the event simply by the activity performed.

Besides the $case\_id$, each case has a mandatory non-null attribute $trace$. Traces are defined over the set $\mathcal{E}^*$ of all finite sequences over $\mathcal{E}$.

The mandatory trace attribute $\hat{c} = \#_{trace}(c)$ of a case is a finite sequence of events $\sigma \in \mathcal{E}^*$ such that each event appears only once in σ. It is generally assumed that traces are non-empty sequences, i.e., they contain at least one event.

With these definitions at hand, we define an event log as set of traces $\mathcal{L}$ such that each event appears at most once in the log. Formally, $set(\hat{c}_1) \cap set(\hat{c}_2) = \emptyset$ for any $\hat{c}_1, \hat{c}_2 \in \mathcal{L}$; where $set(\hat{c})$ denotes the set of events in $\hat{c}$.

Finally, let $\mathcal{A}$ be the set of unique activity names in the log. That is, $\mathcal{A}$ is the set of $\#_{activity}(e)$ in all $e \in \mathcal{L}$.

3.3 Process Discovery

The first task in Process Mining is the *process discovery*. The goal of the process discovery task is to obtain a *process model* from an *event log*. The resulting process model is typically *executable* to allow the reasoning over the identified process. Additionally, since the logs contain the events that actually took place, the resulting process model represents *actual behavior* in the system – rather than an imagined, idealized designed version of the process.

In the plan-recognition and plan-verification tasks that we apply we rely on a three-schemata conceptual model definition of the domain – see the discussion on Chapter 2. The *static* schema captures the entities of the domain, their attributes and relationships. An instance – a grounding - of the concepts defined by the static schema determines a state of the domain. The *dynamic* schema captures the operations that change the state of the world. The generic operator definitions refer to the concepts in the static schema. An operator instance refers to instances of those concepts. Finally, the *behavioral* schema captures the intention and high-level rules of the agents in the domain – including the administrator of the system or automated management rules. This very same paradigm has been shown to be useful for modeling *literary genres* as well environments that support *serious games* domains (CIARLINI, CASANOVA, *et al.*, 2010; GOTTIN, CASANOVA, *et al.*, 2015).

We now describe how each of these schemata can be derived from an event log. Here, we focus on the standard notation for a conceptual model schemata based on Prolog clauses that closely follow (FURTADO e CIARLINI, 2001) – with the exception of one particular abuse of notation in the static schema, the Library of Typical Plans for Process Mining is compatible with the Library of Typical Plans presented in that work with respect to the definition of the conceptual model. We discuss this in Section 4.1.5.

Recall that the default classifier in the literature is $\underline{e} = \#_{activity}(e)$. We require a more complex classifier, however, due to the characteristic of *repeated activities* in the domains we represent. Since each case may execute (or attempt to execute) the same activity multiple times the activity name does uniquely identify an event. With many instances of the same activity in each case's trace, we must use additional event attributes to compose unique names. We identify the events not only by the name of the activity but also by the *timestamp*. This is necessary – and suffices – to uniquely identify events because cases can repeat activities, but never perform the same activity more than once concomitantly. We assume the *timestamp* event attribute is available for the events and define a classifier $\underline{e} = \left(\#_{activity}(e), \#_{timestamp}(e)\right)$.

Recall also that activities may *fail*. We assume a set $\mathcal{V}$ of all the event attributes associated to the results of the activity instances. This set may be empty, but in the formulation below we generally assume that it comprises at least one attribute, since we are dealing with domains with failed attempts at activities. We expect the set of attributes $\mathcal{V}$ to be necessary and sufficient to determine whether the activity instance represented by the event succeeded or failed. In our academic domain example, $\mathcal{V} = \{grade\}$, given by the "grade" column in the log.

In the formalization of the concepts, we assume a domain mapping Q from attributes to atoms in the standard Prolog notation. For example, we map the attribute $timestamp$ to the atom `semester` and the attribute $grade$ to the `grade` atom.

We also assume a mapping A from activity names to atoms. For example, $A[INF1015]$ = `inf1015`, for a discipline "INF1015" in our educational domain. We discuss these and other domain dependent definitions in Section 3.3.4.

3.3.1 Static schema

The static schema definition follows the entity-relationship model with extensions for practical utility.

When relating an event log to a domain, we must determine what are the relevant *entities*. The entity clauses in the static schema, as *meta data*, define the classes of entities that figure in a given domain, whereas the entity instances belonging to these classes constitute the *data* level.

In the definition of event logs we find that all events relate uniquely to a case – each *event* represents the change in a *case*'s story by the execution of an *activity*. Hence, our operations refer to the cases, but also to certain *event attributes*. The arguments in the operations, as described in the next section, refer to the defined entity classes. These event attributes must also be defined as part of the static schema.

In summary, the approach adopted in the present work defines:

- one `entity` clause to represent the case entity, the "class" of all cases;
- one additional clause to represent each of the *event attributes* that are relevant for identifying the events in the domain (that is, for each of the event attributes that are part of the event classifier);
- one additional clause to represent each of the *event attributes* that determine whether an activity instance *succeeds* or *fails* (that is, for each of the event attributes in $\mathcal{V}$ that relate to the result of the activity instance).

Some (or all) of these event attributes may be defined by strictly numerical values. We define these to be represented by `value` clauses, instead of `entity` clauses. This is leveraged later for the Plan Mining approach - when mining typical plans, we intend to effectively capture the interval constraints of these kinds of numerical attributes (see the formal description below and Section 3.3.5).

Relating this to our educational domain, we have:

```
% The case entity
entity(student).
% Additional event-identifying attribute (part of the classifier)
value(semester).
% Activity-instance result attribute (part of set ~V)
value(grade).
```

That is:

- one `entity(student)` clause to represent the students in the domain;
- one `value(semester)` clause to represent the $timestamp$ attribute that is part of the event classifier;
- one `value(grade)` clause to represent the $grade$ event attribute that determines success or failure of the activity instance.

We now describe a general procedure for generating a static schema from the event log information. We formalize it in relation to standard Prolog notation and relate it to our educational domain example.

We define entity clauses of the form:

```
<entity-clause> ::= <entity-functor>(<entity-arguments>)
<entity-functor> ::= entity | quantity
<entity-arguments> ::= <case-class> | <case-class>, <attribute-list>
<attribute-list> ::= <attribute> | <attribute>, <attribute-list>
```

where `<case-class>` is the general identifier – the "class" – of the cases in the domain. In our educational domain, that is the `student` case class; the default `<entity-functor>` is `entity` – we describe the definition of `quantity` entity clauses below.

The optional attributes in the `<attribute-list>` are case attributes that can be used to identify characteristic of the cases. Formally, for each attribute $n \in AN$ such that all cases $c \in \mathcal{C}$ have $\#_n(c) \neq \perp$ we can define an additional argument `<attribute>` for the case entity predicate. In our example, since the only available case attribute is the termination $status$ of the student – and not all students have a termination status – we define no case entity arguments. We discuss more about the case entity arguments below and show an example of a case argument incorporated into the model in Section 3.5.6. A note on this formulation with respect to the formulation in (FURTADO e CIARLINI, 2001): we choose to represent the entity arguments as part of the entity clause, instead of separate attribute clauses. We discuss this in Section 4.1.5.

Notice that our approach assumes that we have a uniform set of cases – they all belong to a same 'class'. This is true in our educational domain: all cases comprise students, with the same attributes and performing the process with similar goals. Assuming a single class of cases present in the log we define a single entity clause. We refer to this single entity as the 'case entity' in the static schema. In our educational domain, the case entity is defined by `entity(student)`.

We now define additional clauses for the event attributes that are part of the classifier $\underline{e}$; and for the event attributes that are in $\mathcal{V}$, i.e., determine the result of the activity instance. In both cases the `<entity-functor>` is `value` if the attribute is strictly numerical, and `entity` otherwise. Also in both cases we assume that the `<entity-arguments>` comprise only the `<case-class>`.

In the general formulation, given the domain classifier $\underline{e} = (\#_{n_1}(e), \#_{n_2}(e), \dots, \#_{n_z}(e))$, each attribute $n_i \in AN$ such that $n_i \neq activity$ and that $\#_{n_i}(e)$ is part of the classifier yields one additional `<entity-clause>` clause where the `<case-class>` is given by $Q[n_i]$. Notice that even though we have the $activity$ event attribute as a part of the classifier, we explicitly disregard it when generating entities from the event classifier attributes. This is because the activity names naturally correspond to the names of the operations in the domain, as discussed in the next section.

Recall that in the example of our educational domain we use a classifier $\underline{e} = \left(\#_{activity}(e), \#_{timestamp}(e)\right)$ such the events are identified both by the discipline code (the $activity$ name) but also by the $timestamp$. With $Q[timestamp] =$ `semester` the attribute $timestamp$ defines one additional `semester` entity clause. Since the $timestamp$ attribute is strictly numerical we define the `<entity-functor>` as `value`.

Finally, we define entities for the event attributes related to the result of the activity instance. Recall the set $\mathcal{V}$ of such event attributes. Similarly as to the event classifier attributes, for each attribute $n \in \mathcal{V}$ we define one additional clause where `<entity-name>` is given by $Q[n]$.

In our educational domain the $result$ attribute is the only member of $\mathcal{V}$ and is also strictly numerical. This follows straightforwardly from the column of the event log from which the data originates – the "grau" column contains numerical values exclusively. With the identity mapping $Q[grade] =$ `grade` we define an entity `value(grade)`.

We now provide some final considerations regarding the static schema and its definition for Process Mining approaches.

Recall from the definition above that entity clauses may contain attributes. This is not the case for the clauses in the above domain example, but the static schema makes allowance for these entity-attributes, and our Library of Typical Plans for Process Mining (Chapter 4) accounts for them in its implementation, guaranteeing full retro-compatibility with the BLIB introduced in (FURTADO e CIARLINI, 2001). We also discuss entity-attributes in the model enhancement task.

Entity-attributes are not the only feature of the Library of Typical Plans for Process Mining that are supported but not intrinsically required for the Process Mining tasks. The static schema also allows hierarchy (`is_a`) and `relationship` clauses representing connections between entity classes. It is hard, however, to extract hierarchical aspects of the static schema directly from the event logs. Hence, we shall assume that hierarchy detection is not required for the Process Mining tasks in our approach.

The same holds both for entity-attributes and relationship definitions. These can be provided by a domain specialist to enrich and guide the plan-recognition and plan-verification tasks but are not strictly required for our intended Process Mining application.

3.3.2 Dynamic schema

The set of operations declared in the dynamic schema restrict what are the changes possible in the domain – the repertoire of actions and events that can take place is entirely represented by the operations. They are defined in terms of their preconditions and effects, following the STRIPS representation. In the following we describe how the dynamic schema can be discovered from an event log. We present one approach for extracting the domain operations from log, given known activities and their dependencies.

The STRIPS formalism separates the processes of theorem proving from those of searching through a space of world models such that “theorem-proving methods are used only within a given world model to answer questions about it concerning which operators are applicable and whether or not goals have been satisfied” (FIKES e NILSSON, 1971). Thus, operations are formulated as conjunctions of positive or negative facts – the preconditions must hold at a certain state for the operation to be applicable; and the effects assert and retract facts as consequences of the execution of the operation. Our planning paradigm conforms to what is described in the literature as *classical planning* (GHALLAB, NAU e TRAVERSO, 2004; GHALLAB, NAU e TRAVERSO, 2016). Formal definitions for the classical representation operators are given in (GHALLAB, NAU e TRAVERSO, 2004, p. 27-33). A planning operator is a triple *o =(name(o), preconditions(o), effects(o)).*

A formal specification of a behavior schema is given in (CIARLINI, CASANOVA, *et al.*, 2010). We define `added` clauses, corresponding to the *effects*, and `precond` clauses, corresponding to the *preconditions*. We define an `operation` clause that is composed by an *operator signature,* corresponding to the operation *name,* and an accompanying *operator frame* that restricts the objects that can be used to instantiate operators as actions (idem, (p. 33)). Notice that in the most general formulation the effects of operations may also include `deleted` clauses. As will become clear, in the simplest configuration that is necessary for Process Mining these are not strictly necessary for the activity operations. That is the case in the initial modeling of our domain. We discuss how model extensions could require negative effects from operation in Section 3.5. Notice also that the planning mechanisms described herein account for deleted facts as effects of operations, as evidenced by the compatibility of the Library of Typical Plans for Process Mining with previous works (see Chapter 4).

In the dynamic schema in our conceptual model does not account for external, or exogenous, events. We assume the repertoire of operations to be the only ways through which changes happen in the world. In the Process Mining context, we are especially concerned with the execution of activities by the cases – hence we assume it is possible to extract from an event log the operations that are the actions of agents in the domain.

We have previously discussed that the *disciplines* correspond to the *activities* in the process. Hence, in our educational domain example, we expect to extract from the log a dynamic schema definition of the operations performed only by students. In defining the dynamic schema, our goal is to represent the *traces* in the event log as *plots* in our conceptual model definition. Recall that a trace is a finite sequence of events associated to a case. A *plot*, in our definition, comprises a sequence of operations related to an entity. We discuss this further in Section 3.3.5.

Other than the general activities in the domain, we define operations relating to the successful and unsuccessful termination of *cases*. We relate this to the definition of processes with a *knock-out* structure. Van der Aalst defines a *knock-out process* as “a business situation where for each case a pre-specified set of tasks needs to be executed” where “the processing of a task stops immediately if in one of the tasks a reason for rejection is detected” (2001, p. 452). Notice that in this definition it takes but one activity-failure event for the case to be classified as a failure. In our case (of an educational domain), we generalize that definition to include other, more complex rules for a *knock-out*. Hence, we must also define operations relating to the *drop out* and *completion* of cases.

The definition of the operator signatures and frames, as well as the preconditions and effects, leverages the fact that the activities in the domain are known. Typically, there is available information, in some readily available data source, that allows the automatic definition of the dependencies between activities in standard Prolog notation. That is the case of our educational domain example, in which we obtain from the available documentation the ‘pre-requisite’ relationships between disciplines. Otherwise, the dependencies between activities should be provided manually, from domain knowledge (see Section 3.3.4).

Alternatively, we could resort to techniques described in the literature of (planning) *domain knowledge acquisition* (see Chapter 2). Our assumption of modeling the activities in terms of intertask dependencies also resembles the discovery procedures for graphical models discussed at length in the Process Mining literature. Many algorithms, like the seminal *Alpha* algorithm (VAN DER AALST, 2011) and the *HeuristicsMiner* (WEIJTERS, VAN DER AALST e DE MEDEIROS, 2006; WEIJTERS e RIBEIRO, 2011), for example, rely on the patterns of orderings between activities in the event log to determine dependency relations (the *footprint matrix* and *dependency graph*, respectively). There are several issues with these approaches related to the completeness of the log, noise in the log and frequency thresholds, for example, but under certain assumptions these have been demonstrated to be able to reliably extract the dependencies between activities in the domain. We posit that such approaches could be used to determine preconditions and effects as well. In fact, there are works in the literature that extract planning domain operators from graphical models such as the Petri nets (see Section 2.3).

For the definition of the operator signatures and frames, and the preconditions and effects, we leverage `activity` clauses obtained from the domain describing the *intertask dependencies*. Recall that we consider the possibility of *multiple dependencies* for activities. This means some activities may have zero, one, or several *alternative* sets of dependencies. For example, in our educational domain:

- discipline INF1406 does not have any prerequisites;
- discipline INF1022 requires both INF1007 and INF1009; and
- discipline INF1007 requires *either* INF1005 *or* INF1025.

In Prolog notation:

```
activity( inf1406, [] ).
activity( inf1022, [ [inf1007, inf1009 ]).
activity( inf1007, [ [inf1005] ,
                     [inf1025]).
```

Formally, for each activity $a \in \mathcal{A}$ we define one activity clause of the format:

```
<activity-clause> ::= activity(<operation-name>,<activity-dependencies>).
<activity-dependencies> ::= [] | [<activity-dependencies>]
<dependency-setlist>::= <dependency-set> | <dependency-set>,<dependency-setlist>
<dependency-set> ::= <activity-dep> | <activity-dep>, <dependency-set>
```

The `<operation-name>` is an atom that corresponds to the activity name, given by the mapping $\mathcal{A}[a]$. The `<activity-dep>` terms are of the same format.

3.3.2.1 Operator signatures and frames

3.3.2.1.1 Activity operations

We start by defining the operation clauses – comprising the *operator signature* and the *operator frame*, and how they can be derived from the event log and the activity clauses.

The operator signature corresponds to the *operator name*, in the classical planning representation definitions. The operator frame restricts the values of the arguments in the operator signature to certain entities defined in the static schema

– it defines *typed variables*, in the terminology of planning domain analysis (GHALLAB, NAU e TRAVERSO, 2004, p. 534).

We define the first argument of each operation to be related to the case entity since activity instances are performed by a single case (recall that *every single event in the log is linked to exactly one case* in the log). Recall that the operations are the *activities* of the domain – hence, we determine the remaining parameters of the operations based on the relevant *event attributes* in the event log. There are two kinds of relevant parameters:

- Event-identifying attributes: those that *identify* the event, (that is, those that compose the event classifier $\underline{e}$ for the domain);
- Activity instance result attributes: those that determine the outcome of the activity instance captured by the event (that, is those that belong to the set $\mathcal{V}$).

These are reminiscent, and in fact defined, in terms of the entities defined in the static schema.

The most general definition of an operation corresponding to a domain activity is given as follows:

```
operation( activity(Student,Discipline,Timestamp,Grade),    %operator signature
           [student/a, discipline/o, semester/in, grade/with] %operator frame
         ).
```

In this definition the discipline and timestamp arguments are given by the classifier, and the grade argument is related to the result of the activity instance. We adopt a different notation, however, for convenience purposes. Since the plots of the student comprise mostly discipline activities (only disciplines and at most one case termination operation per case), we 'highlight' the discipline code and use it as the *functor* of the activity operation signatures, in an abuse of the notation. This makes the plots (and later the mined plans) easier to read, with minor impact on the implementation of the mechanisms. The method and the algorithms implemented as part of the Library of Typical Plans for Process Mining could be adapted to the more general representation with reasonable effort.

An example of an operation in the adopted notation is given:

```
operation( inf1005(Student,Timestamp,Grade),    %operator signature
           [student/o, semester/in, grade/with] %operator frame
         ).
```

This example denotes that an event of discipline INF1005 performed by a *student* that in the *semester*, obtaining a resulting *grade*. A grounding of the operator signature `inf1005('X',2,85)` means "Discipline INF1005 was completed by student `X` in semester 2, she completes it with grade 85".

In the definition of the operator signature the arguments are unbound variables. *The relation between arguments and entities is comprehended by the operator frame.* In our educational domain, the `student` entity relates to the first argument because it is the case entity. The `semester` and `grade` entities compose the operator frame because they identify the event and determine the outcome of the activity instance, respectively.

A general procedure for determining the `operation(<op-signature>, <op-frame>)` clause is as follows. We exemplify the steps of the procedure with a restricted example of our educational domain.

Recall the set $\mathcal{A}$ of unique activity names in the log. For each $a \in \mathcal{A}$ – and, therefore, for each `activity` clause - we define one `operation` clause of the format:

```
<operation-clause> ::= operation(<op-signature>,<op-frame>)
```

That is, the operation is defined by an *operator signature* and a matching *operator frame*:

```
<op-signature> ::= <operation-name>(<op-case>, <op-arglist>)
<op-arglist> ::= <op-id-arglist> |  <op-id-arglist>, <op-res-arglist>
<op-id-arglist> ::= <op-id-arg> | <op-id-arg>, <op-id-arglist>
<op-res-arglist> ::= <op-res-arg> | <op-res-arg>, <op-res-arglist>

<op-frame> ::= [<case-entity>/o, <frame-arglist>]
<frame-arglist> ::= <frame-id-arglist> | <frame-id-arglist>, <frame-res-arglist>
<frame-id-arglist> ::= <frame-id-arg> | <frame-id-arg>, <frame-id-arglist>
<frame-res-arglist> ::= <op-res-arg> | <frame-res-arg>, <frame-res-arglist>
<frame-id-arg> ::= <frame-entity>/<frame-role>
<frame–res-arg> ::= <frame-entity>/<frame-role>
```

The operation signature is a term with the `<operation-name>` functor, and with a set of arguments given by `<op-case>,<op-arglist>`. The operation frame is a list `[<case-entity>/o, <frame-arglist>]`.

The arguments of the `<op-signature>` always include as the first argument a variable to represent the case entity in the static schema, the `<op-case>`. This matches the first element `<case-entity>/o` in the `<op-frame>`. That is: the first element of the frame is the case entity and the role o, denoting that the case is the object of the operation (this relates to the representation of the operation instances in natural language – we choose to represent the case as the 'object' in our adopted notation, rather than the 'agent' in the most general formulation).

In the educational domain, for each discipline we define one operator signature. That is, the activity name is the discipline name (via the mapping $A[a]$) and the `student` case entity is the first element of the operator frame. At this stage, an informal (and incomplete) representation of the operation clauses for the activities figuring in Table 1 is:

```
operation( inf1413(Student, … ), [ student/o, …  ] ).
operation( inf1636(Student, … ), [ student/o, …  ] ).
operation( inf1406(Student, … ), [ student/o, …  ] ).
operation( fis1033(Student, … ), [ student/o, …  ] ).
```

We now define the `<op-arglist>` and the `<frame-arglist>` that complete the '…' in the informal representations of the operation signature and operation frames, respectively.

Each of the arguments in the list `<op-arglist>` is a variable, matching an element in the `<frame-arglist>` that relates to one `entity` (possibly a `value`) defined in the *static schema*. These are the agents or values involved in the action that the operation represents. Each of the arguments in the operator signature matches an element of the form entity/role in the same relative position in the `<op-frame>`.

The `role` is used to compose natural language representations of plots and does not directly impact the planning algorithms. We discuss this in Section 3.3.4.

The `<op-arglist>` is comprised by `<op-id-arglist>` and, optionally, by `<op-res-arglist>`. Respectively, the `<frame-arglist>` is comprised by `<frame-id-arglist>` and, optionally, by the `<frame-res-arglist>`.

The `<op-id-arglist>` arguments and their matching `<frame-id-arglist>` arguments are derived from the event attributes that, along with the activity name, identify the event. The `<op-res-arglist>` arguments and their matching `<frame-res-arglist>` arguments are derived from the event attributes that, along with the activity name, relate to the results of the activity instance in the event.

Given the domain classifier $\underline{e} = (\#_{n_1}(e), \#_{n_2}(e), \dots, \#_{n_z}(e))$, each attribute $n_i \in AN$ such that $n_i \neq activity$ and that $\#_{n_i}(e)$ is part of the classifier yields one variable in the `<op-id-arglist>`. For each such attribute n_i we define one element `<classifier-entity>/<role>` term in the operator frame, where `<classifier-entity>` is given by $Q[n_i]$ and `<role>` must be defined with domain knowledge. Since a similar procedure is used to determine entities in the static schema, we can safely assume that for each such argument, there exists one corresponding entity such that the `<classifier-entity>/<role>` term that can be added to the operation frame.

In the example of our educational domain, the events are identified both by the discipline code (the *activity* name) but also by the $timestamp$. That is, we use a classifier $\underline{e} = \left(\#_{activity}(e), \#_{timestamp}(e)\right)$. With $Q[timestamp] =$ `semester` the attribute $timestamp$ yields one element `semester/in` in the operator frame. A corresponding variable argument is added to the operator signature. The role `in` for the semesters is defined with domain knowledge. Hence, at this point we have (informally represented) operation clauses like:

```
operation( inf1413(Student, Semester, … ), [ student/o, semester/in, …  ] ).
operation( inf1636(Student, Semester, … ), [ student/o, semester/in, …  ] ).
operation( inf1406(Student, Semester, … ), [ student/o, semester/in, …  ] ).
operation( fis1033(Student, Semester, … ), [ student/o, semester/in, …  ] ).
```

Recall the set $\mathcal{V}$ of event attribute names that determine the *result* of the activity – success or failure. Each attribute $n \in \mathcal{V}$ yields one element in the `<op-res-arglist>`. Similarly, as above, for each such attribute n we define one element `<result-entity>/<role>` term in the operator frame, where `<result-entity>` is given by $Q[n_i]$ and the `<role>` is given by domain knowledge (see again Section 3.3.4).

In our domain example the set $\mathcal{V} = \{result\}$, and $Q[result] =$ `grade`, so only the `grade` entity is added to the operator frame. A corresponding variable argument is added to the operator signature. We obtain the complete operation clauses:

```
operation( inf1413(Student, Semester, Grade),
           [ student/o, semester/in, grade/with ]
         ).
operation( inf1636(Student, Semester, Grade),
           [ student/o, semester/in, grade/with ]
         ).
operation( inf1406(Student, Semester, Grade),
           [ student/o, semester/in, grade/with ]
         ).
operation( fis1033(Student, Semester, Grade),
           [ student/o, semester/in, grade/with ]
         ).
```

In practice, we don’t define one operation clause for each activity. Rather, we define a generic `operation` predicate that represents all the activity operations in the domain, relying on the `activity` clauses.

3.3.2.1.2 Case termination operations

The other operations defined in the dynamic schema are related to the successful *completion* of cases and the unsuccessful termination of cases (*knockout*). We call these the *case termination operations*.

We require case termination operations to distinguish *partial traces* from *knockout cases* in the domain. Hence, we assume at least one case termination operation representing the successful *completion* of the case, and at least one termination operation representing the halting or aborting of the event.

As examples in our educational domain, we define case termination operations:

```
operation( graduate(Student, Semester), [student/o, semester/in] ).
operation( dropout(Student, Semester), [student/o, semester/in] ).
```

They are defined in the same format – an `operation` clause comprising an operation signature `<op-signature>` and an operation frame `<op-frame>`. For ease of representation we allow these operations to have to have domain-defined `<operation-name>` terms (hence `graduate` and `dropout`). These names are also used to determine the effects of these case termination operations. The choice of names is briefly discussed in Section 3.3.4.

In the definition of case termination operations, we do not include the event attributes that relate to the results of activities. Hence, the `<op-arglist>` in the `<op-signature>` comprises only the `<op-id-arglist>`, referring to the event attributes in the event classifier. Similarly, the `<frame-arglist>` in the `<op-frame>` comprises only the `<frame-id-arglist>`.

In our educational domain example, this means that the case termination operations include one argument for the semesters (part of the event classifier) but do not contain one argument not for the grade (which relates to the results obtained in the disciplines).

An important aspect of the case termination operations is that they may not be explicitly available in the log. The termination information may be missing or associated to another data source. In this discussion, we assume that the case attribute $\#_{status}(c)$ is available, from which we can determine the appropriate termination operation for the case's trace. Not all students have a termination $status$ value, which impeded the use of that attribute as a case entity argument. However, the missing information does not stop us from leveraging it to define the case termination operations – we simply *do not* generate case termination operations for ongoing traces.

Furthermore, there may be *several* ways in which a case may terminate, and these ways must be accounted for by one case termination operation each. Hence, there may be a strong domain-dependent component in the definition of these operations. Here we assume the simpler case where only one of which is defined. This is also discussed in Section 3.3.4.

3.3.2.2 Preconditions and effects

In order to complete the repertoire of operations we must define their preconditions and effects (also called in literature *post-conditions*).

In the definition of preconditions and effects we leverage the fact that, for the purposes of Process Mining, we are mostly interested in the *dependency relations between activities*. In general, we aim to represent the *intertask dependencies* and the *multiple dependencies* between activities in terms of preconditions.

The preconditions of the operations will mainly refer to the successful completion of other operations. Conversely, the effects of operations will typically comprise the results (success or failure) of the activity instances. Hence, we leverage the activity clauses and the explicit dependencies they contain to define the preconditions and effects of the operations. We discuss how more general preconditions and effects are incorporated to our approach for process mining in Section 3.5. That discussion also considers the use of negative effects given by `deleted` clauses.

Recall that in our intended application domains the activities may fail. The effects and preconditions of the operations will be given in terms of the *results of the activity instance* represented by the operation. Thus, we define *success* and *failure* clauses `<op-success>` and `<op-failure>`. We will then compose the `added` and `precond` clauses in relation to clauses in this format:

```
<op-success> ::= <success-functor>(<op-case>,<activity-attr>,<op-id-arglist>).
<op-failure> ::= <failure-functor>(<op-case>,<activity-attr>,<op-id-arglist>).
```

Notice that, for ease of representation, we allow the success and failure terms to have domain-defined functors (`<success-functor>` and `<failure-functor>`). In this discussion we will use `success` and `failure`, respectively - in examples in the next Sections we may use `app` and `rep`, respectively, but this will be pointed out in those examples (see Section 3.3.4).

Similarly to the definition of the operation signature, the first argument of the `success` and `failure` clauses is always a variable related to the case in the static schema, given by `<op-case>`, as informally represented below:

```
success(Case, … ).
failure(Case, … ) .
```

The second argument of the `success` and `failure` clauses is a variable that relates to the activity name:

```
success(Student, Discipline, … ).
failure(Student, Discipline, … ) .
```

The `<op-id-arglist>` arguments correspond are similarly composed as in the definition of the operator signature: given the domain classifier $\underline{e} = (\#_{n_1}(e), \#_{n_2}(e), \dots, \#_{n_z}(e))$, each attribute $n_i \in AN$ such that $n_i \neq activity$ and that $\#_{n_i}(e)$ is part of the classifier will yield one element in the `<op-id-arglist>`.

In our educational domain, besides the $activity$, the events are also identified by the $timestamp$ attribute. Hence:

```
success(Student, Discipline, Semester ).
failure(Student, Discipline, Semester ) .
```

Notice that unlike the definition of the operation signature, *there is no frame for the success and failure clauses* – the first argument `Student` is a variable, e.g, and *it will only refer to instances of the case entity in the static schema due to the* *context in which the success and failure clauses are used*, within the `added` and `precond` clauses. The same is true for the other arguments. Hence why we are free to change the variable names and choose to use `Student`, `Discipline` and `Semester` – as per the terminology of the domain.

3.3.2.2.1 Activity operations

A general formulation for the preconditions and effects clauses given known intertask dependency relations is given below in Prolog notation. We rely on the representation of activities and their dependencies in the `activity` clauses, and in the `success` and `failure` clauses defined above. Since we have one `operation` clause for each $a \in \mathcal{A}$, we will be able to define the `precond` clause(s) for all operations based on the `<activity-dependencies>` of the corresponding activity.

We define two clauses to represent the effects of each activity - one to reflect the *success* and the other to reflect the *failure* of the activity instance. For each activity a, we define two `added` clauses:

```
added( <op-signature>, <op-success> ):- <conditions>.
added( <op-signature>, <op-failure> ).
```

The term `<op-success>` or `<op-failure>` will match the `<op-signature>` with respect to the case variable and the event-identifying attributes. We will have, for example, for discipline INF1413:

```
added( inf1413(Student, Semester, Grade), success(Student,inf1413,Semester)):-
      passing_grade(Grade).
added( inf1413(Student, Semester, Grade), failure(Student,inf1413,Semester)).
```

The first `added` clause is a rule whose body `<conditions>` comprises domain-dependent goals that determine whether the operation succeeds, typically related to the event attributes in $\mathcal{V}$. We provide further discussion in Section 3.3.4. In our example above we have defined a predicate `passing_grade/1` that succeeds when the `Grade` obtained by the student is sufficient, according to domain rules.

We now define the preconditions of the operations, leveraging the activity dependencies of the activity corresponding to the operation. Preconditions are given by `precond` clauses.

We will determine zero, one, or many `precond` clauses for each operation; depending on the sets of dependencies of the corresponding `activity` clause. If the activity is without dependencies, with `<activity-dependencies>` as an empty list, we define no `precond` clauses for that operation. That is the case of INF1406 in our domain (see above). For operations whose matching activity have a one or more sets of dependencies we generate one `precond` clause *for each set of dependencies*.

In the case of activity INF1022 in our example, we generate a single `precond` clause for the requirement of both INF1077 and INF1009:

```
precond( inf1022(Student, SemesterA, _Grade),     %operator signature
         ( success(Student, inf1077, SemesterB), SemesterB < SemesterA),
           success(Student, inf1009, SemesterC), SemesterC < SemesterA) )
).
```

For activity INF1007, we will generate two `precond` clauses – one to represent the pre-requisite INF1005 and the other to represent the *alternative* pre-requisite INF1025:

```
precond( inf1007(Student, SemesterA, _Grade),
         ( success(Student, inf1005, SemesterB), SemesterB < SemesterA).

precond( inf1007(Student, SemesterA, _Grade),
         ( success(Student, inf1025, SemesterB), SemesterB < SemesterA).
```

Notice in the examples above that we add dependencies between the 'semester' of the discipline and its pre-requisite(s). This is a particular instance of *additional dependencies* that is relevant to our domain. We provide a general formalization below, and further discussion in Section 3.3.4. Notice also that the 'Grade' in the operation signature is forcibly a singleton variable. Event attributes that relates to the results of the operation intuitively have no effect in the preconditions of that operation.

Formally, for activity $a \in \mathcal{A}$ we obtain the corresponding `<activity-dependencies>` from the activity clause `activity(A, <activity-dependencies>)`, where `A` is the atom given by $A[a]$. Let o be the operation defined from activity a (recall that we have defined one `operation` clause for each $a \in \mathcal{A}$ also). In o, the atom `A` is the functor of the operation signature.

Each d in the `<activity-dependencies>` of activity a defines a *precondition conjunction* π, such that if π holds the operation is applicable in that situation. Let Π be the set of all alternative precondition conjunctions of o. We will define one `precond` clause of operation o for each $\pi \in \Pi$:

```
precond(<op-signature>, (<op-preconds>)).
```

Recall that each element d in the `<activity-dependencies>` is a *list* of activity names. In our example of INF1022, the `<activity-dependencies>` contains a single list `[inf1077, inf1009]`. Thus we generate a single precondition conjunction for the single `precond` clause of the operation representing INF1022 (as exemplified above). Conversely, for INF1007, it contains two lists: `[inf1005]` and `[inf1025]` – and so we generate two `precond` clauses for the operation representing that discipline (also exemplified above).

Each list d of activity names defines a precondition conjunction π given in Prolog notation as a term `<op-preconds>`:

```
<op-preconds> ::= <op-prec> | <op-prec>, <op-preconds>

<op-prec> ::= <op-success>, <op-id-depslist>
<op-id-depslist> ::= <op-id-dep> | <op-id-dep>, <op-id-depslist>
```

The precondition conjunction is given by one or many terms `<op-prec>`, depending on the number of activities in d. Each such term is composed of a *success clause* in conjunction with *additional dependencies over the event-identifying attributes*.

In the example of INF1022, the single element d = `[inf1077, inf1009]` defines an `<op-preconds>` conjunction of two `<op-prec>` terms, one referring to `inf1077` and one referring to `inf1009`.

We determine each `<op-success>` term in the `<op-prec>` such that the `<op-case>` matches the `<op-case>` case in the operation signature. Notice in the examples above that the `Student` variable is the first argument in the operation signature and in each of the `success` clauses.

We determine the `<activity-attr>` in the success clause with the atoms in d. In the examples above, this is the name of the pre-requisite disciplines.

Finally, each argument of the `<op-id-arglist>` is a new variable. In the examples above, the semesters in which the student must have completed the pre-requisite discipline(s) are intuitively different.

The terms `<op-id-depslist>` define the additional dependencies between the event identifying attributes (in the example, the precedence requirements between the “semester” of the discipline and the pre-requisite(s)). This dependency depends on semantics of the attribute in the domain but will typically relate to the event identifying attributes. We discuss this briefly in Section 3.3.4.

3.3.2.2.2 Case termination operations

Finally, other than the operations defined by activities we have the *case termination operations*, related to the completion and knockout of cases. These case termination operations will typically have simple effect – in our example, asserting that the operation took place and that the case is over.

Recall that we allow domain-specific names for these operations - for our educational domain we defined operations `graduate` and `dropout`, for example. We similarly assume that for each such operation a unary `<case-termination>` is defined:

```
<case-termination>::= <termination>(<op-case>).
```

We define for the `graduate` and `dropout` operations the terms `graduated` and `dropped`, respectively. We define the effects of the case termination operations with the respective case termination terms:

```
added(graduate(Student, Semester), graduated(Student)).
deleted(graduate(Student, Semester), student(Student)).
added(dropout(Student, Semester), dropped(Student)).
deleted(dropout(Student, Semester), student(Student)).
```

The case termination operations may have more complex *preconditions*, however. For example, in our educational domain, one way in which students drop from the program is the compulsory dismissal due to to repeated failures in the same activity.

```
precond( dropout(Student, Semester),
         ( failure(Student, Discipline, SemesterA),
           failure(Student, Discipline, SemesterB),
           failure(Student, Discipline, SemesterC),
           failure(Student, Discipline, SemesterD),
           failure(Student, Discipline, SemesterE),
           SemesterA #< SemesterB, SemesterB #< SemesterC,
           SemesterC #< SemesterD, SemesterD #< SemesterE ) ).
```

In the example above we represent the precondition to the dropout case termination operation as the student failing several times in the same discipline. This is a simplified version (a more relaxed requirement, even) of the rule in the actual domain, as discussed in the Appendix. As we'll discuss in future examples even this relaxed configuration is typically not satisfied in the domain.

We also define simplified criteria for graduation:

```
precond( graduate(Student,Sem),
    ( % 1st semester
      app(Student,fis1033,_), app(Student,fis1034,_), app(Student,inf1005,_),
      app(Student,mat1161,_), app(Student,mat1200,_),

     % 2nd semester
      app(Student,cre1100,_), app(Student,inf1007,_), app(Student,inf1009,_),
      app(Student,inf1008,_), app(Student,inf1403,_), app(Student,mat1162,_),

     % 3rd semester
      app(Student,cre0700,_), app(Student,eng1029,_), app(Student,inf1010,_),
      app(Student,inf1012,_), app(Student,inf1018,_), app(Student,mat1154,_),

     % 4th semester
      app(Student,ele1030,_), app(Student,inf1301,_), app(Student,inf1383,_),
      app(Student,inf1626,_), app(Student,inf1019,_), app(Student,inf1631,_),

     % 5th semestre
      app(Student,cre1141,_), app(Student,inf1011,_), app(Student,inf1377,_),
      app(Student,inf1608,_), app(Student,inf1636,_), app(Student,inf1721,_),
      app(Student,inf1715,_),

     % 6th semester
      app(Student,cre1172,_), app(Student,fil0300,_), app(Student,inf1013,_),
      app(Student,inf1016,_), app(Student,inf1640,_), app(Student,inf1771,_),

     % 7th semester
      app(Student,inf1612,_), app(Student,inf1950,_),
      app(Student,inf1014,_), app(Student,inf1413,_),

     % 8th semester
      app(Student,inf1015,_), app(Student,let0310,_),
      app(Student,inf1920,_), app(Student,inf1951,_) ) ).
```

The above represents a simplified set of the actual requirements for graduation in the domain, comprising the set of 'required' disciplines. In actuality, the criteria for graduating comprises the obtention of credits in predetermined disciplines and groups of disciplines in the curricula (see the Appendix). We revisit this discussion in Section 3.5.1. Nonetheless, we will consider the preconditions above for the examples in the following examples. These examples illustrate how the pre-conditions for case termination operations depend on the domain rules –we discuss more about the `precond` clauses for these operations in Section 3.3.4.

3.3.3 Behavioral schema

The behavioral schema defines a set of *goal-inference rules* that capture the behavior and the motivations of agents in the domain. The definition of goal-inference rules is paramount to the plan-recognition algorithms.

Goal-inference rules are defined in terms of an *agent*, a *situation*, and a *goal*: "a goal-inference rule has, as antecedent, some situation which, if observed at a database state, will arouse in a given agent the impulse to act in order to reach some goal" (FURTADO e CIARLINI, 2001). Goal-inference rules are defined in our standard Prolog notation as follows:

```
gi_rule(<rule-id>, <rule-agent>, <rule-situation>, <rule-goal>).
```

Where `<rule-id>` is a unique rule identifier. The `<rule-agent>` is, by default, the case entity that the rule refers to. The `<rule-situation>` and `<rule-goal>` are conjunctions of terms that represent, respectively, the *current* state of a case in which a rule is applicable and the *goal* state to be sought by the plan-recognition procedure. These will typically comprehend terms defined in the static schema, plus the `success` and `failure` terms that are preconditions and effects of the operations in the domain. In these terms, the `<rule-agent>` will always unify with the `<op-case>`.

In a Process Mining application there are two sets of rules: those known a priori, which can be modelled with domain knowledge, and those that are *unknown* and cannot be determined by domain specialists. We assume that the set of goal-inference rules known a priori will include at least one rule, representing a *normative pattern*. A normative pattern determines a "*de jure model*" that "specifies how things should be done or handled" (VAN DER AALST, 2011, p. 243). That is, a (typically idealized) version of the process as defined by the domain specialists and administrators. This definition is intrinsically tied to the domain knowledge.

In our educational domain, for example, we define custom rules for representing the *recommended order of disciplines* that the program suggests students should follow, plus others. Some details are given in the Appendix. The following rule exemplifies the *normative pattern* of the students, from the initial enrollment, performing with success the exact order of recommended disciplines:

```
gi_rule( 2,
               student(Student),
               ( student(Student), not success(Student, _, _), not
rep(Student,_, _) ),
               ( not rep(Student,_, _),
       success(Student, inf1403, S1), success(Student, inf1005, S1),
       success(Student, mat1200, S1), success(Student, mat1161, S1),
       success(Student, fis1033, S1), success(Student, fis1034, S1),
       success(Student, inf1008, S2), success(Student, inf1009, S2),
       success(Student, inf1007, S2), success(Student, mat1162, S2),
       success(Student, cre1100, S2), success(Student, inf1012, S3),
       success(Student, inf1010, S3), success(Student, inf1018, S3),
       success(Student, eng1029, S3), success(Student, mat1154, S3),
       success(Student, cre0700, S3), success(Student, inf1383, S4),
       success(Student, inf1301, S4), success(Student, inf1631, S4),
       success(Student, inf1626, S4), success(Student, inf1019, S4),
       success(Student, eng1400, S4), success(Student, inf1377, S5),
       success(Student, inf1636, S5), success(Student, inf1721, S5),
       success(Student, inf1011, S5), success(Student, inf1715, S5),
       success(Student, inf1608, S5), success(Student, cre1141, S5),
       success(Student, inf1013, S6), success(Student, fil0300, S6),
       success(Student, inf1771, S6), success(Student, inf1016, S6),
       success(Student, inf1640, S6), success(Student, cre1172, S6),
       success(Student, inf1413, S7), success(Student, inf0310, S7),
       success(Student, inf1950, S7), success(Student, inf1014, S7),
       success(Student, inf1015, S8), success(Student, let0310, S8),
       success(Student, inf1951, S8), success(Student, inf1920, S8),
       S1 #< S2, S2 #< S3, S3 #< S4, S4 #< S5, S5 #< S6, S6 #< S7, S7 #< S8) ).
```

The following rule, for example, determines that students at the beginning of the program should aim to complete the recommended disciplines for the first semester, without any failures, in their first semester:

```
gi_rule( 3,
        % Agent
        student(Student),
        % Situation
        ( student(Student)
          not success(Student, _, _), not failure(Student,_, _) ),
        % Goal
        ( not failure(Student,_, _),
          success(Student, inf1403, 1), success(Student, inf1005, 1),
          success(Student, mat1200, 1), success(Student, mat1161, 1),
          success(Student, fis1033, 1), success(Student, fis1034, 1) ) ).
```

Notice that the goal-inference rules determine a higher-level “de jure model” than a strict graphical model. There may be countless combinations of activities that the cases perform to achieve a goal. The mining of typical plans over the normative pattern rules will indicate not only *if* but also *how* the cases perform that pattern.

The actual plans followed by cases correspond to the *“de facto models*” (VAN DER AALST, 2011, p. 243) that capture the reality of the domain. Discovering these from the event log is the core concern of the process discovery task. We will discuss how our approach deals with the discovery of plans from goal-inference rules in Section 3.3.5.

The analysis of how the plots of the cases conform with (or deviate from) the normative pattern rules may highlight issues with the patterns. We discuss these kinds of analysis, with examples of discovered issues, in Section 3.4. The discovery of these issues may shift the interest of the analysis towards other patterns that are hypothesized to be more frequent, more relevant, or more interesting. In Section 3.5 we will explore how new rules can be (semi-)automatically derived from normative pattern rules. We discuss the implementation of the mechanism in our Library of Typical Plans for Process Mining in Chapter 4.

3.3.4 Three-Schemata domain dependent definitions

In this section we recapitulate and discuss in more detail some domain dependent aspects of the Process Discovery of the three-schemata conceptual model from event logs.

The first domain dependent definition is the set of attributes that identify the event. Recall that the most basic classifier for our approach is $\underline{e} = (\#_{activity}(e), \#_{timestamp}(e))$. In our educational domain, we use that default classifier. If there are additional event attributes that are necessary to uniquely identify the event, they must also be included in the classifier. We discuss the effects of these additional attributes in the following.

Including additional attributes in the event classifier incurs in additional `entity` clauses in the static schema. If these attributes are strictly numerical, they are defined by `value` clauses instead. In the definition of the dynamic schema, each additional attribute also incurs in one more argument in the operation signatures and, correspondingly, in the operation frames. Finally, the `<op-success>` and `<op-failure>` terms in the definition of preconditions and effects will also have one more corresponding argument.

Another domain dependent definition is the set of attributes $\mathcal{V}$ that relates to event attributes that represent the result of the activity instances. If the domain does not contain activities that may fail, this set may be empty. In our educational domain we assume that the result of the activity is dictated by the attribute $grade$, from the column ‘grau’ in the event log. Similarly, as for the additional attributes in the classifier, additional attributes in $\mathcal{V}$ will incur in additional `entity` (or `value`) clauses in the static schema. They also similarly incur in additional arguments in the operation signature, operation frame and the `op-success>` and `<op-failure>` terms.

Recall that we allow for domain-defined functors for the `op-success>` and `<op-failure>` terms. Throughout Section 3.3 we have used success and failure, for clarity, but in the remainder of this thesis we use the domain-defined `app` and `rep`, respectively. We recall this decision in the examples.

We also define mappings from names in the event log to atoms in the Prolog notation. Mapping Q maps from attribute names, comprehending both event and case attributes, to Prolog atoms. That is, for each attribute $n \in AN$ the mapping $Q[n]$ yields a unique atom.

We also define mapping A from activity names to atoms. For any $a \in \mathcal{A}$ the mapping $A[a]$ yields a lowercase atom that can be used as a functor. Since each activity $a \in \mathcal{A}$ has a unique name we trust it is straightforward to define this mapping such that $A[a]$ is uniquely related to a.

A domain-dependent definition that typically will not require domain knowledge input from an expert is the definition of which entities are given by value entities, in the static schema. It typically suffices to verify that these are derived from numerical-valued columns in the log. The same is not true for the definition of roles, in the operation frames. The role of the case entity can typically be default to the *object* of the operation. We could define default roles for typical event attributes, such as the timestamp. Other attributes will invariably require input from a domain expert. This is not a limitation to the automatic discovery of the dynamic schema, however, since the roles are only used for the natural language generation aspect and are not required for the plan-recognition and plan-verification tasks.

Also in the definition of operations, there are domain dependent definitions for the *case termination* operations. We allow for domain-defined names for these operations. In our educational domain, we use `graduate` and `dropout`, respectively, for the mandatory completion and knockout operations.

Recall also that we define one case termination operation for each way in which the case may terminate. That is, if there are several ways in which the case may be knocked-out, we could define one knockout operation for each of those ways. Even though we initially choose to represent only one general knockout operation `dropout` for our educational domain, we could define one operation to represent a student that is remove from the program due to repeating a same discipline five times; and another to represent a student that exceeds the maximum amount of semesters before graduating; and another to represent students that transfer from PUC to another higher-education institutions. We will discuss more about this when leveraging information in the log to increment the process model with respect to case termination operations in Section 3.5.

In the definition of the effects of operations, we determined a predicate `added` with an arbitrary body `<conditions>` to represent the success cases. The definition of the conditions will be dependent on domain knowledge as well. These conditions should typically be defined *over the set of event attributes* $\mathcal{V}$, however, since those are the event attributes related to the results of operations in the domain.

In our educational domain example, for example, we define a predicate `passing_grade/1` such that determines if the student is approved or fails the discipline based on the grade obtained. This simple requirement is a simplification of the rules for approval in disciplines in the academic program.

We also define domain dependent dependencies `<op-id-depslist>` in the preconditions of the operations. These will require input from domain knowledge but are typically related to the event identifying attributes in the event classifier $\underline{e}$. In our domain, for the $timestamp$ attribute, we presume a straightforward precedence relationship.

In our educational domain example, this is the case of dependencies over the semester of the disciplines – that is, a student must have been approved in the pre-requisite discipline(s) *before* enrolling in a discipline. In Chapter 4 we discuss how we adapt the plan-verification and plan-recognition mechanisms to consider the $timestamp$ information.

Another domain-dependent aspect of the conceptual model is that "pre-conditions and effects are usually tuned in a combined fashion, aiming at the enforcement of integrity constraints" of the domain (FURTADO e CIARLINI, 2001). We could add, for example, to each preconditions clause an additional fact to ensure that a student that has been approved in a discipline cannot perform it again, as such:

```
precond( inf1022(Student, SemesterA, _Grade),          %operator signature
           not success(Student, inf1022, SemesterB),    % cannot repeat
        (...)                                           %remaining definitions
).
```

We choose not to do so for simplicity of representation – we have previously verified that this does not happen in any of our example cases.

The case termination operations assert that the case is over. We define unary predicates with the same name as the case termination operations to indicate that those operations were executed. For example, in our educational domain, having defined the `graduate` and `dropout` operations, we define the corresponding case termination terms `graduated/1` and `dropped/1`.

The preconditions of case termination operations are also domain-dependent definitions. Unlike the conditions in the activity operations, they will not relate to activity result event attributes. They will instead typically relate to *rules of the process*, to *exogenous events*, and to the success and failures of sequences of activities and to time limits. The latter two are easier to represent. The former two are even mode domain dependent.

For example, in our educational domain, we determine domain rules for the *graduate* and *dropout* events with simplifications with respect to the actual domain rules. The consequences of the simplifications we assumed in the modeling of these preconditions are discussed, and partially amended, in Section 3.5. Finally, the discovery of additional goal-inference rules – which relates to the discovery of the Behavioral Schema – is also dependent on the domain. We discuss some general methods that can automatically extract new rules in Section 3.5, but the design of these methods is performed with the needs and characteristics of the domain in mind.

3.3.5 Mining of Typical Plans

In this section we discuss how plan-recognition allows the *mining of typical plans* and how that relates to the Process Discovery task. We leverage and extend the approach defined in (FURTADO e CIARLINI, 2001). The plan mining approach employs the goal-inference rules to seek for relevant patterns in the plots. Here, we give an overview of the plan mining approach and how it relates to the Process Discovery task. In Chapter 4 we describe the Library of Typical Plans for Process Mining and how it implements the approach, with examples in a simplified educational domain.

The plan mining method relies on the goal-inference rules to determine how agents (the cases) proceed from determined situations towards relevant goals. The mechanism of mining of typical plans relates to the Process Discovery task: it collects plans that are *actually executed* by cases in the domain, as opposed to ideal plans, dictated by domain specialist(s). The typical plans executed correspond to the *de facto models* of what is *actually* performed by the cases, regardless of the process envisioned by the domain administrators. In our educational domain we aim to understand what are the sequences of disciplines that students are performing.

The plan mining algorithm mines plans from *plots*. We obtain plots from the cases' *traces*.

Recall Table 1 (p. 37). Van der Aalst projects the case and the classifier attributes (only the activity name, in his case) to obtain a table of traces (VAN DER AALST, 2011, p. 14). In our educational example domain, with the classifier including both the activity name and the timestamp, we have traces comprising tuples, as can be seen in Table 2.

Table 2 A trace.

Case id	**Trace**
2368	`fis1033(2368,1), fis1034(2368,1), inf1005(2368,1), inf1403(2368,1), mat1161(2368,1), mat1200(2368,1),` `...,` `inf1021(2368,14), inf1408(2368,14), inf1640(2368,14), inf1951(2368,14)`
2501	`(cre1100, 1), (fis1033, 1), (fis1034, 1), (inf1009, 1), (inf1012, 1), (inf1403, 1), ..., (inf1190, 6), (inf1406, 6), (let1296, 6)`
...	...

We suppress parts of the traces for ease of representation. Recall that $\hat{c}$ is the trace of the case c.

In our conceptual model, we compose *plots* from the traces of cases as predicates `plot/2` of the format:

```
plot( <case-id>, [s0, <plot-opslist>] ).
```

```
<plot-opslist> ::= [<op-signature>, <plot-opslist>]
```

This represents the trace of case `<case-id>` as a sequence of operations given by `<plot-opslist>`. The operations that compose the plot are ordered, respecting the timestamp. All operations with the same timestamp are simultaneous in the plot. The special operation annotation `s0` that starts each plot represents the initial state of the case - the state of the world at the beginning of the trace, before any operations take place.

Each element of the list is an operation signature, *grounded* to represent the activity *instance* performed by the case in an event. Recall that the event attributes in $\mathcal{V}$ determine the result of the activity instance and are part of the operation signatures. Hence, our *plots* correspond to the *traces*, with the additional attributes related to the activity instance results.

An example of a plot corresponding to the trace of case of student 2368 in Table 2 is shown below, in informal notation that suppresses a part of the plot:

```
plot(2368, [s0,
                fis1033(2368,1,71), fis1034(2368,1,84),
                inf1005(2368,1,81), inf1403(2368,1,43),
                mat1161(2368,1,64), mat1200(2368,1,63),
                …,
                inf1021(2368,14,81), inf1408(2368,14,97),
                inf1640(2368,14,68), inf1951(2368,14,100)]).
```

It is straightforward to obtain the operation signature including the event attributes in $\mathcal{V}$ because the traces contain information to *uniquely* identify the events. In the example above, there is only *one* instance of student 2368 performing the discipline fis1033 in semester 1 – hence, it is straightforward to obtain from the event log that his grade was (unambiguously) 71.

Notice that the plots composed from cases traces will refer only to the case. This is not a restriction of our conceptual model – the Library of Typical Plans for Process Mining retains the ability to mine plans from plots involving actions of multiple agents, as described in (FURTADO e CIARLINI, 2001). This is discussed in Chapter 4. The special operation annotation `s0` refers to the initial state – the state of the work before the plot. In (FURTADO e CIARLINI, 2001) this could configure complex scenarios, of on-going domains, involving multiple agents. Since our plots refer to a single case, and all cases start in a similar manner, it suffices to define in the initial state only the entity related to the plot. We define a clause `initial_database` of the format:

```
initial_database( <case-id>, <entity-clause> ).
```

Where the `<entity-clause>` is the clause defining the case entity. In our educational domain, the `initial_database` of each case is given by the student clause:

```
initial_database( 2367, student(2367) ).
```

In Section 3.5 we extend the entity clause definition to account for case attributes. This will also reflect in the definition of the `initial_database` clauses.

The goal of the plan mining approach is to obtain *plans* from the plots, following the indications of behavior given by the goal-inference rules. The approach relies on a plan-recognition method that identifies a part of a plot as potential plan.

Notice that the plan, if it exists, is a *grounded* plan, composed by grounded operations $[s0, o_1, o_2, \dots, o_k]$ and with the support of a single case. Because all the operations in each plot refer to the same case, all operations o_i in the plan refer to the same agent (they all have the same argument, a case instance). The Library of Typical Plans for Process Mining retains the capabilities of (FURTADO e CIARLINI, 2001) for dealing with general domains, which includes a (backwards-chaining) mechanism for extracting from the plan only the operations that are necessary for the plan's execution. This is also discussed in Chapter 4.

If the resulting plan is composed of a single operation already defined in the dynamic schema, we record it as a *simple* plan. However, if the plan comprises a partially ordered set of multiple operations, we define a *complex* plan that is a *composite* operation. This operation is defined as part of the mining process– hence, we generate an operation signature and operation frame to represent it. We discuss the automatic generation of operations in Section 3.5.5.

The mining of typical plans will perform the plan-recognition described above for *all* cases in the domain. Hence, several alternative plans can be found. For both simple plans and complex plans of composite operations we attempt to *generalize* the plans. We resort to two methods of generalization. The first one is based on the *most specific generalization* of plans with similar structure (FURTADO, 1992). The second one is based on the definition of *generic* operations.

The most specific generalization of plans is performed when two (or more) of the plans recognized for a same rule are *similar*. In that case, we can store a general version of the plan that accounts for both (or several) plans. We update the cases associated to the plan accordingly, to reflect the plan's increased support.

Assume, for example that for rule R we recognize plan p_1 from case c_1. Assume that we also recognize plan p_2 from case c_2. If it is possible to determine that p_1 and p_2 are *similar*, we store a generalized plan $p_{1,2}$ that supports cases c_1 and c_2. If we later identify an additional plot p_3 from case c_3 that is similar to $p_{1,2}$, we produce a plan $p_{1,2,3}$ that supports all three cases.

Notice that the method above relies on the definition of what are *similar* plans. We consider two plans *similar* if they are similar in structure – with respect to the operations that compose them, and the order requirements between those operations – and if it is possible to *unify the lifted versions* of those plans. This includes the enforcement of *co-designation* and *non-co-designation* of variables in the generalized operations.

We now consider the case when multiple plans are recognized for a rule but are not similar. Assume the scenario in which, for rule R, we recognize plan p_1 from case c_1 and plan p_2 from case c_2 and we determine that p_1 and p_2 are *not* similar. This happens when the plans have a different number of operations, are composed by a different set of operations, or fail the co-designation/non-co-designation restrictions. In this scenario, we generate a new operation $g_{1,2}$ to represent the *generic* operation of performing either p_1 or p_2. Generic operations can represent both simple and composite operations.

Hence, a plan is ultimately defined as an *operation* – either simple (a single operation in the dynamic schema) or a complex operation (a composite operation or a generic operation, defined during the mining process). We represent a plan as follows:

```
% A plan is either simple or complex
<plan> ::= <simple-plan> | <complex-plan>
```

```
% Simple plans are given by a single operation – defined in the dynamic schema –
% plus plan cases and constraints (see below)
<simple_plan> ::= simple:(single:<op-signature>,
                          cases:[<case-list>],
                          constraints:[<ctr-list>])

%Complex plans are given a complex operation - either a composite or a generic operation
<complex_plan> ::= complex:(<composite-op>) | complex:(<generic-op>)

% A composite operation is defined by components and the order dependencies between them,
% plus plan cases and constraints (see below)
<composite-op> ::= composite:<op-signature>,
                   frame:<op-frame>,
                   components:[ <comps-list> ],
                   dependencies:[ <deps-list ],
                   cases:[ <case-list> ],
                   constraints:[<ctr-list>]

<comps-list> ::= <comp-def> | <comp-def>,<comps-list>
<comp-def> ::= <fid>:<op-signature>
<deps-list> ::= <fid>-<fid> | <fid>-<fid>, <deps-list>

% A generic operation is defined by its specializations
% plus cases (see below)
<generic-op> ::= generic:<op-signature>,
                 frame:<op-frame>,
                 specializations:[<specs-list>],
                cases:[ <case-list> ]
<specs-list> ::= <op-signature> | <op-signature>, <specs-list>

% Plan definitions: lists of cases and lists of constraints, respectively
<plan-defs> ::= cases:[<case-list>], constraints:[<ctr-list>]
<case-list> ::= <case-id> | <case-id>,<case-list>
<ctr-list> ::= <ctr> | <ctr>,<ctr-list>
```

Notice that single operations are defined in the dynamic schema and therefore do not contain the frame of the operation.

For the simple plans and the complex plans given by composite operations we mine the *constraints* over the arguments that are given by `value` clauses in the static schema. In the current implementation of the Library of Typical Plans for Process Mining, these constraints are given as *ranges* of numerical values.

The complex plans given by generic operations are simplified to refer only to the *case* – with no other arguments – and hold no explicit *constraints*. In the representation above and in all other representations of plans we show an *explicit* list of constraints for the variables. As we discuss in Section 4.1.2, the manipulations of the constrained variables are all reliant on annotated variables via the CLP(FD) library. We define a mechanism for extracting an explicit representation of the constraints to store them as part of the Library Index structure, assert them as facts in the Prolog database. This is discussed in Section 4.1.3.

All plans contain the lists of *cases* that perform that plan. This is later used to determine the support of the plan for decision making and analysis. This can later be used for the conformance checking with respect to 'how frequently' a plan aligns with another, or with a normative pattern. It also enables the decision mining, based on characteristics of the cases, as discussed in Section 3.5.

The complex operations are defined in the mining process, and as such the plans will also contain a definition of the operation frame. The simple operations correspond to the operations defined in the static schema – hence only the operation signature suffices.

For composite operations we follow the definitions in (FURTADO e CIARLINI, 2001): We define a list of `components (`representing part-of links) tagged with atoms `<fid>` such as `f1,f2,…,` and a list of `dependencies` whose elements `<fid>-<fid>`, such as `f1-f2,` denote order dependency requirements. These dependencies are determined based on the satisfaction of the preconditions and effects. The list of dependencies will not include transitive dependencies. Eg, if the list contains `f1-f2` and `f2-f3` the element `f1-f3` is omitted – it can be derived by transitivity from the other dependency requirements. This standard format is leveraged for comparing the similarity of composite operations, since similar operations must have similar order dependencies of their components.

Below, we provide examples of a simple plan, a complex plan given by a composite operation, and a complex plan given by a generic operation in our educational domain. These are arbitrary plans, mined from an arbitrary goal-inference rule, just for the purposes of demonstration.

```
simple:(single: ctc1002(2368,1,87),
       cases:[2368],
       constraints:[]).
```

The above plan means that student 2368 reached the goal *g* from situation *s* by performing only the discipline CTC1002, in his first semester, obtaining a grade of 87. The plan definitions relate that 2368 is the only case following this plan – hence why it is a *grounded* operation.

Assuming that the mining process later finds that another student, 2390, also reaches the goal with a similar plan, it will *generalize* this simple plan. That is, instead of keeping both plans, it will consider the following generalized plan:

```
simple:(single: ctc1002(S,1,A),
       cases:[2368, 2390],
       constraints:[ (A in 81..87)]).
```

The resulting plan is given by an operation signature that is not *grounded* - in this case, with respect to the first argument (the case id) nor the third argument (the grade). Since students 2368 and 2390 had different grades in the discipline, the *range of grades* is kept as a constraint over the variable `A`, representing both their grades.

Below we see an example of a complex plan given by a *composite* operation.

```
complex:( composite:      c1(S),
          frame:          [student/o],
          components:     [f1: mat1161(S,1,A),
                           f2: mat1162(S,B,C) ],
          dependencies:   [f1-f2],
           cases:          [2371, 2387, 2388],
           constraints:    [(A in 81..89),
                           (B in 2..3),
                           (C in 61..98) ] ).
```

The composite operation `c1` is defined with only the case id as a single argument, and a matching operator frame. This plan represents that students 2371, 2387 and 2388 reached the goal from the situation by passing discipline MAT1161 in the first semesters, with a grade varying between 81 and 89. They also passed discipline MAT1162 either in the second or third semester (as given by the interval constraint over variable `B`). The order dependencies between the execution of these operations are explicitly given by the `dependencies` term.

Assume that both plans above are recognized as plans *for the same rule*. Since they are not similar plans – as per our definition above – we add a *generic* operation to represent both as *alternative* plans:

```
complex:( generic:        g1(S),
          frame:          [ student/o],
          specializations:[ ctc1002(S,1,A),
                            c1(S) ],
           cases:          [2368, 2371, 2387, 2388, 2390]).
```

The generic operation g1(S) means that there are two distinct plans to reach the goal – one is the plan given by the single operation of discipline CTC1002 in the first semester; and the other is the complex plan given by the composite operation `c1`. Notice that in the definition of the generic operations the arguments are variables – we follow the design decision from (FURTADO e CIARLINI, 2001), which relieves us from having to propagate changes in variables in view of future detections of similar plans.

As a concrete example in our educational domain, recall the goal-inference rule described in Section 3.3.3. Notice that we use the `app` and `rep` functors for the `op-success>` and `<op-failure>` terms in place of `success` and `failure`, respectively, as previously discussed in Section 3.3.4.

```
gi_rule( 3, student(Student),                                 % Agent
       ( student(Student),
         not app(Student, _, _), not rep(Student,_, _) ),       % Situation
       ( not rep(Student,_, _),                                % Goal
         app(Student, inf1403, 1), app(Student, inf1005, 1),
         app(Student, mat1200, 1), app(Student, mat1161, 1),
         app(Student, fis1033, 1), app(Student, fis1034, 1) ) ).
```

That rule represents the goal of students who just enrolled in the program to complete the recommended disciplines for the first semester in their first academic term. For a set of students in the domain we obtain the following complex plan:

```
complex : (composite : c1(A),
           frame: [student/o],
           components: [ f1 : fis1033(A,1,B),
                         f2 : fis1034(A,1,C),
                         f3 : inf1005(A,1,D),
                         f4 : inf1403(A,1,E),
                         f5 : mat1161(A,1,F),
                         f6 : mat1200(A,1,G)
                       ],
           dependencies: [],
           cases: [2388,2387,2371],
           constraints: [ (B in 68 .. 90),
                          (C in 81 .. 100),
                          (D in 93 .. 99),
                          (E in 69 .. 85),
                          (F in 81 .. 89),
                          (G in 68 .. 72)
                        ]
          ).
```

This means that only three case 2388, 2387 and 2371 – out of the set of 23 cases considered - achieved this goal following this plan. In the next section we discuss how the cases recorded in the plans are used to reason about the *support* of plans. In Chapter 4 we discuss the implementation of the Plan Mining mechanism in the Library of Typical Plans for Process Mining.

As an additional example, consider the goal-inference rule with a `<rule-id>` identifier 8-1:

```
gi_rule( 8-1, student(Student),
           ( student(Student),
             rep(Student, D1, S),
             not app(Student, D2, S)
            ),
            graduated(Student) ).
```

This rule captures the plans of graduated students from a semester with failed disciplines and no successful disciplines until graduation. For this rule, in the same set of students we obtain the following plans:

```
complex : ( generic: g1(A),
            frame: [student/o],
            specializations: [c2(A),c1(A)],
            cases : [2389,2387,2371] ),

complex : ( composite: c2(2389),
            frame: [student/o],
            components:[...],                     %suppressed for the example
            dependencies:[...],                   %suppressed for the example
            cases: [2389],
            constraints: []),

complex : ( composite: c1(B),
            frame: [student/o],
            components: [f1:inf1951(B,C,D), f2:graduate(B,14)],
            dependencies: [f1-f2],
            cases : [2387,2371],
            constraints : [ (C in 13..14), (D in 75..100)])
```

Notice that the plans of students 2371 and 2387 are captured in a single composite operation via most specific generalization. We suppress the components and dependencies of the plan c2 of student 2389 due to its size (it contains 30 operations). Given both plans relate to the same goal-inference rule, we compose a generic operation in which each of the composite operations figure as specializations. This mechanism is described in Section 4.1.2.

3.4 Conformance Checking

The second main task of Process Mining approaches is the *conformance checking*. This task relates a *process model* (an executable model) to an *event log*. The goal is to find the commonalities and discrepancies between the modeled behavior and the behavior in the event log. In this section we discuss how plan-verification over the typical plans obtained from the mining described in Section 3.3.5 can be used to achieve the goals of conformance checking.

In the literature the technique is described in terms of *token replay* (as opposed to the *play-out*, or simulation, and *play-in*, process discovery). That terminology is derived from the *tokens* in the graphical models such as Petri Nets. We use the plan-verification mechanisms to *replay* the plots according to the operations in our conceptual model.

The literature also describes how conformance checking can be used to compare an event log against a *descriptive* model, or against a *normative* model. In the first case, the goal is to identify discrepancies that indicate how the process model must be improved to adequately capture that behavior. In the case of *normative* models, the deviations indicate desirable (or undesirable) deviations from the process as it is originally designed.

Accordingly, in this section we will discuss the conformance checking in two levels. The first level is the conformance checking of the conceptual model discovered from the event log (and domain knowledge), as described in the Section 3.3. We show how the planning approach allows us to reason about the discrepancies in insightful ways. We draw correspondences between this level and the conformance checking via *replay*, as well as the task of *business alignment* or *auditing*. This is discussed in depth in Section 3.4.1. The consequences of the auditing may motivate the model repair as a model enhancement task (see Section 3.5.1).

The second level consists of the conformance checking of the typical plans, discovered via the plan mining techniques discussed in Section 3.3.5 and implemented in the Library of Typical Plans for Process Mining. We show how the reasoning of the *support* of the plans – and related metrics - in relation to the rules can provide insights of the conformance of the model with respect to the normative patterns. We relate the support to the *fitness* of the model. This is discussed in Section 3.4.2.

The conformance checking approaches described here are directly tied to the approaches described in the literature. However, the conceptual model formalism and the planning mechanisms lend themselves to other approaches that could also achieve the goals of conformance checking. As stated in (FURTADO, 1992) "formalisms are useful in *verification* tasks even when they have to be complemented by the expert's intuition in tasks that involve *discovery*". We could leverage the approaches for formal verification to find deviations and discrepancies between the event logs, the domain specifications, and the plans.

One such approach is the *model checking* approach (GOTTIN, 2013). Model checking is a technique for determining whether a property holds in a formally specified system – typically, a transition system or a *Kripke structure* (BROWNE, CLARKE e GRÜMBERG, 1988). Model checking algorithms explore the state space that is defined by the system specification to determine the logical restrictions are satisfied. The properties to be verified are typically temporal, that is, logical formulae that express constraints about the properties of the events over time. Despite the problems related to state-space explosion, there are practical applications of model checking for planning domains, including in partially observable domains (DÓRIA, CIARLINI e ANDREATTA, 2008). In our conformance checking approach we could leverage that model checking is typically applicable to planning domains to verify hypotheses about complex properties of the system over time.

The conformance checking approaches described below also provide insights for the definition of additional goal-inference rules, both for the analysis of the domain as well as for 'model repair'. We recapitulate this discussion in Section 3.5.

3.4.1 Plot replay

In this section we discuss how the planning techniques allow for the conformance checking of the process models as represented by the conceptual modeling discipline. Van der Aalst (2011, p. 193) discusses a "concrete technique for quantifying conformance and diagnosing non-conformance" on graphical models based on *token replay*. This technique is performed re-executing the traces in the log, comparing them to the possible firings in a Petri Net model. In a first approach the replay is stopped upon detecting a problem (i.e. a case event that cannot be captured by the model). That, however, hides the issues in later parts of the model, especially when many of the cases cannot be 'parsed' completely. The approach proposed is then to *continue* replaying the trace on the model while recording the problems. In a Petri Net model this accounts for keeping a record of 'missing tokens' (VAN DER AALST, 2011, p. 195) .

We implement a *replay* mechanism based on plan-verification, and used the operations defined in the dynamic schema to detect the preconditions of operations that are not met.

Like in the token replay approach, we do not stop replaying a plot upon finding a problem. Rather, in a kind of counter-factual analysis, we *allow* the failed preconditions to hold, recording that that is so, and propagating the effects of the failed operation forwards.

For the replay of plots, we rely on a forward-chaining of operations – that is, a mechanism similar to *simulation* – in which the preconditions of each operation are tested for satisfiability given the previous operations in the plot.

Like in van der Aalst's token replay, we don't stop replaying the trace upon encountering a problem. Rather, we collect the *discrepant operations* (along with their preconditions) and continue checking the remainder of the plot. At the end of the conformance checking via replay, we obtain *all of the discrepant operations* (along with the respective sets of preconditions of the discrepant operations, for further analysis).

As an example, in our educational domain, we consider the case of student 2368 (from Table 2 and previous examples). The plot consists of 61 operations. The process finds almost all of them to be conformant. However, one of the disciplines performed in the student's last semester is found to be discrepant. The process reports the following:

```
% The allowed operations
[]
%The failed operations
[inf1640(2368,14,68)]
%The preconditions that could not be met
[ inf1640(2368,14,68),
        [ [ (app(2368,eng1400,A), A #< 14, app(2368,inf1019,B), B #< 14,)
            (app(2368,eng1400,C), C #< 14, app(2368,inf1316,D), D #< 14)
        ] ]
]
```

Meaning that the *noncoformance* of the discipline INF1640 does not affect further operations. We can also see that neither of the two alternative preconditions for the operation were met. Upon examining the student's plot, we verify that indeed he did not perform the discipline ENG1400, required in both alternative preconditions.

And also, via the recursive invocation in which we allow the failed operation:

```
% The allowed operations
[inf1640(2368,14,68)]
%The failed operations
[ ]
%The preconditions that could not be met
[ ]
```

Meaning that if we disregard that nonconformance, no additional problems with the plot can be found.

Notice that in this example, student's 2368 plot *is a partial trace*. It does not include a case termination operation and represents that the student is still performing the process. Hence, the conformance checking mechanism can be used within the context of *online process mining*. This characteristic of the conformance checking task is highlighted by van der Aalst : "*all events in a business process can be evaluated and this can be done while the process is still running*. The availability of log data and advanced process mining techniques enable new forms of auditing (…). Process mining in general, and conformance checking in particular, provide the means to do so." (VAN DER AALST, 2011, p. 194).

As another example, consider the case of student 2371.

```
% The allowed operations
[]
```

```
%The failed operations
[ inf1640(2371,5,72),  psi1849(2371,7,100),
  psi1847(2371,8,100), inf1416(2371,9,62), graduate(2371,14) ]
%The preconditions that could not be met
[ inf1640(2371,5,72),
        [ [ app(2371, eng1400, A), A #< 5,
            app(2371, inf1019, B), B #< 5
          ],
          [ app(2371, eng1400, C), C #< 5,
            app(2371, inf1316, D), D #< 5
          ]
        ],
  psi1849(2371,7,100),
        [ [ app(2371, psi1848, E), E#<7 ]
        ],
  psi1847(2371,8,100),
        [ [ app(2371, psi1840, F), F#<8 ],
          [ app(2371, psi1848, G), G#<8 ]
        ],
  (...) %Suppressed for ease of representation
]
```

In this case, several operations are found to be discrepant. We find that typically disciplines of other departments are discrepant. For example, PSI1849 and PSI1847 above are disciplines offered by another department, as indicated by the code 'PSI' (see the Appendix). Since the students in the program typically only perform a small number of such disciplines, it is common to find that their pre-requisites, which tend to be for disciplines of the same area, are not met.

In this example above we suppress part of the sets of preconditions that could not be met, for clarity. In the examples below they are suppressed entirely. In one of the recursive invocations (we're skipping some of the results for ease of representation) of the conformance checking replay, we obtain:

```
% The allowed operations
[ psi1849(2371,7,100)]
%The failed operations
[ inf1640(2371,5,72), psi1847(2371,8,100),
  inf1416(2371,9,62), graduate(2371,14) ]
%The preconditions that could not be met are suppressed in this example
```

That is, by allowing the operation PSI1849 (as if it were conformant), we still verify that the other operations are discrepant. In another of the recursive invocations we obtain:

```
% The allowed operations
[ inf1640(2371,5,72) ]
%The failed operations
[ psi1849(2371,7,100), psi1847(2371,8,100), graduate(2371,14) ]
%The preconditions that could not be met are suppressed in this example
```

Notice that by allowing INF1640, the discipline INF1416 would no longer be considered discrepant. This means that the execution of INF1640 satisfies the missing requirements of INF1416. Furthermore, consider the results of yet another recursive invocation in this case:

```
% The allowed operations
[ inf1640(2371,5,72), psi1849(2371,7,100), psi1847(2371,8,100)]
%The failed operations
[ graduate(2371,14) ]
%The preconditions that could not be met are suppressed in this example
```

This example illustrates how several operations can be ‘allowed’ for the conformance checking replay to go on. In this particular example, we observe that *even if we allowed the three discrepant operations*, the student still should not have been able to graduate, according to the requirements determined in the case termination operation preconditions for this example.

This motivates the model repair, as we will discuss in Section 3.5.1. We provide additional discussion on the implementation of the conformance checking replay method in 4.2.

3.4.2 Model fitness

One of the four main quality criteria defined for the quality of process models is the *fitness*, related to the capacity of the model of representing the behavior observed in the domain – that is, the behavior reflected in the event log. The fitness of a model measures “the proportion of behavior in the event log possible according to the model” (VAN DER AALST, 2011, p. 194).

The goal of this verification is to assert the reliability or confidence of the model. In our case, we relate this to the verification of how the plots conform to the goal-inference rules and, by extension, to each *typical plan*, since the goal-inference rules implicitly define a set of plans that reach a goal from a certain situation. We will rely on metrics related to the *support* of the rules (and plans), deriving the concepts from association rule mining.

3.4.2.1 Support and confidence of goal-inference rules

We will leverage the information of the *cases* that are identified for each of the typical plans to determine the *support* of the plan in the domain, as well as other metrics of interest, drawing from the literature in *frequent itemsets* and association rules.

Recall that an association rule is defined as an implication of the form X⇒Y, where the antecedent X and consequent Y (also called the left-hand-side, or LSH, and right-hand-side, RSH, respectively) are *itemsets*. The support(X∪Y) (also represented as supp(XUY)) is the ratio between the number of cases that satisfy the rule (with both the antecedent and consequent) over the total items in the domain.

We define a notion of support for our goal-inference rules. Our goal-inference rules are given by an agent a, a situation s, and a goal g. Considering the situation s as the antecedent and the goal g as consequent, we define the support($s \Rightarrow g$) of the rule as the ratio between the number of cases that follow the plans in the rule and the total number of cases in the domain.

Recall $\mathcal{C}$ is the set of all cases in the domain. Let k_r be the number of cases that perform a plan in rule $r = (a, s, g)$ – that is, k_r is the number of cases that reach the goal g after the reaching the situation s via any of the plans in the goal-inference rule r. That number is the count of the union of all cases found in all plans of the goal-inference rule. Then, the support($s \Rightarrow g$) = $\frac{k_r}{|\mathcal{C}|}$.

Figure 1 shows a visual representation. The representation introduced in this Figure will be used for later examples.

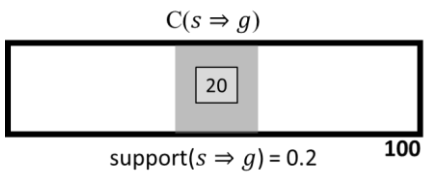


Figure 1 - A visual interpretation of the support.

In this figure the rectangle box comprises the set of 100 cases in the domain. The dark portion represents the cases that perform a plan of the rule – in this case, 20. The number of those cases divided by the total number of cases in the domain straightforwardly, yields the support($s \Rightarrow g$) = 0.2. The support is also called the coverage in the literature – it represents the proportion of the domain *covered* by the set. In our case, one-fifth of the students in the domain perform a plan in this rule.

Recall the goal-inference with `<rule-id>` 2 described in Section 4.3 – with the approval in all required disciplines in the recommended order and no failures. We find in the typical plan mining that *no students ever perform a plan* under those restrictions. Hence, the support($s \Rightarrow g$) of rule 2 is zero.

Recall the goal-inference 3 originally described in that same section and again in the example of Section 3.3.5, as well as the typical plan mined using that rule, also from the same example.

```
gi_rule( 3, student(Student),                                     % Agent
       ( student(Student),
         not app(Student, _, _), not rep(Student,_, _) ),         % Situation
       ( not rep(Student,_, _),                                   % Goal
         app(Student, inf1403, 1), app(Student, inf1005, 1),
         app(Student, mat1200, 1), app(Student, mat1161, 1),
         app(Student, fis1033, 1), app(Student, fis1034, 1) ) ).

% A composite plan (simplified)
complex : (composite : c1(A), (...),
           cases: [2388,2387,2371], (... )  ).
```

The plan describes how three students (2388, 2387, and 2371) are the only ones that, from the initial enrollment, succeed in the disciplines recommended for the first semesters. These are the students whose plot reach the goal's situation and the goal's rule. Assuming in this and the following examples that there are 23 cases in the set of students considered. Thus, the support($s \Rightarrow g$) of rule 3 is 0.1304.

The support of the rule is a metric of how frequently agents have followed plans for that rule in the domain. The support($s \Rightarrow g$) captures the relevance of the rule 3 above nicely because the situation s of that rule – being a student at the beginning of the academic program - *is met by all students in the domain*.

Consider the case of goal-inference with `<rule-id>` 8-1, also from Section 3.3.5:

```
gi_rule( 8-1, student(Student),
          ( student(Student), rep(Student, D1, S), not app(Student, D2, S) ),
           graduated(Student) ).
```

Consider the mined plan's cases, in a short informal representation:

```
complex : ( generic:  g1(A), (...),      cases : [2389,2387,2371] ),
complex : ( composite: c2(2389), (...), cases:  [2389], (...) ),
complex : ( composite: c1(A), (...),    cases : [2387,2371], (...) ).
```

The union of cases that performs plans g1, c2 and c1 is {2389, 2387,2371}. Hence, the support of this rule is also 0.1304.

However, in this case, the rule's *situation is not something that happens for all cases*. It only applies to students that had a semester without any successful disciplines (and are seeking to graduate). In this case, it is interesting to know how often the plan is performed *given that the situation that defines the rule holds*.

We relate this to the concept of *confidence* in the literature of association rule mining. The confidence(X⇒Y) can be interpreted as an "estimate of the probability P(Y|X), the probability of finding the RHS of the rule in transactions under the condition that these transactions also contain the LHS" (HORNIK, GRÜN e HAHSLER, 2005). It is computed as the ratio support(X∪Y)/support(X), where support(X) is the number of cases that contain the antecedent over the total items in the domain.

In our approach, let C(s) be the set of cases in the domain whose plot's reach the situation s – but not necessarily the goal g. Let also support(s)= $\frac{C(s)}{|\mathcal{C}|}$.

We define the confidence($s \Rightarrow g$) of the rule as support($s \Rightarrow g$)/support(s). This gives an indication of how many of the students that *could* follow a plan in that rule *successfully* do so.

Figure 2 shows a visual representation.

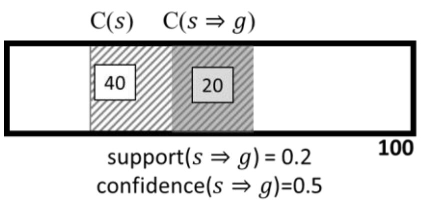


Figure 2 A visual representation of the confidence of a goal-inference rule.

In this figure again the box represents the domain with 100 cases. The hashed portion represents the set of cases C(s). As before, the dark area represents the set cases that perform a plan of the rule, from the situation to the goal. The latter set is intuitively a subset of the first. The confidence of the rule is proportional to the size of the latter set in contrast to the first.

We implement a mechanism for computing the support of a situation s, described in Section 4.2.2. There are eleven cases in the domain that reach the situation of rule 8-1 – that is, 11 out of the 23 students perform a semester without any successful disciplines. With the support($s \Rightarrow g$) = 0.1304 and the support(s) = 0.4782 we have the confidence($s \Rightarrow g$) = 0.2727. This is a low value of confidence – we cannot rely on the situation as a predictor of the goal.

Notice again that the confidence metric for the rules, like rule 3, whose situation is reached by the plots of all cases, is uninteresting. In those cases, the support(s)=1 and, by definition, the confidence($s \Rightarrow g$) will be equal to the support($s \Rightarrow g$).

3.4.2.2 Support and confidence of plans

So far, we have only discussed the support and confidence of goal-inference rules. We can also compute the above metrics per plan. For goal-inference rules that yield several plans, we typically define a generic operation - the support and confidence of those rule will typically correspond to the support and confidence of that plan given by a generic operation. With the information of the support(s) at hand, we define a recursive procedure that computes the support and confidence of each of the plans. For the example of goal-inference 8-1, we have:

```
Support of Rule 8-1:
        support:    0.13043478260869565
        confidence: 0.2727272727272727
        >g1(A):
                support:    0.13043478260869565
                confidence: 0.2727272727272727
        >c2(2389):
                support:    0.043478260869565216
                confidence: 0.0909090909090909
        >c1(A):
                support:    0.08695652173913043
                confidence: 0.1818181818181818
```

Again the low support scores mean that the plans are not *frequent* in the domain – very few students perform the plans from *situation* to *goal*. The low confidence values mean that most of the students that *could* perform the plan – that reach the goal-inference rule's situation – do not reach the goal. Take the plan given by operation `c2`: we have a *support* of 0.0434 and a *confidence* of 0.0909. This means that plan is only performed by 1 out of the 23 students, and that 10 out of the 11 students that could – that were in the situation of having a semester with no successful activities – did not follow this plan for graduating.

Operation `c1` has double the support and confidence of `c2`; it was a plan followed by two students (as opposed to a single one). Given that generic operations are composed to represent the union of the plans mined from a goal-inference rule, it is straightforward to realize that the support and confidence of the generic operations will comprise the sum of the support of its specializations. This is a rather simple example, but illustrates how different plans within the same rule can have different support and confidence metrics.

We further discuss the implementation of the mechanisms for computing support and confidence of plans in the Library of Typical Plans for Process Mining in Section 4.2.2.

The reporting of support and confidence of plans also applies to simple plans. Consider the goal-inference rule 9 below:

```
gi_rule( 9, student(Student),
       ( student(Student), rep(Student, _D1, 1)),
        rep(Student, _D2, 2)
).
```

This rule captures the behavior of students that start the program failing one discipline and proceed to fail in a discipline in the following semester. For this rule, we obtain the plans:

```
complex : (generic :         g6(2384),
           frame :           [student/o],
           specializations : [ mat1157(_,_,_), c19(2384), inf1008(2384,_,_),
                               fis1033(2384,_,_), inf1009(2384,_,_) ],
           cases :           [2390,2385,2384,2377,2376,2370,2368]

simple : (single :       mat1157(2390,2,20),
           cases :       [2390],
           constraints : []),

complex : (composite :   c1(A),
           frame :       [student/o],
           components :  [f1:inf1005(A,1,B),f2:inf1007(A,2,C)],
           dependencies :[f1-f2],
           cases :       [2385,2384],
           constraints : [(B in 69..71), (C in 11..17)]),

simple : (single :       inf1008(_,2,_),
          cases :        [2382,2381,2379,2377],
          constraints :  []),

simple : (single :       fis1033(2376,2,0),
          cases :        [2376],
          constraints :  []),

simple : (single :      inf1009(_,2,_),
          cases :       [2383,2370,2368],
          constraints : [])
```

And the following interest metrics:

```
Support of Rule 9:
        support:    0.4782608695652174
        confidence: 0.7333333333333334
        >mat1157(2390,2,20):
                support:    0.043478260869565216
                confidence: 0.06666666666666667
        >c1(A):
                support:    0.08695652173913043
                confidence: 0.13333333333333333
        >inf1008(A,2,B):
                support:    0.17391304347826086
                confidence: 0.26666666666666666
        >fis1033(2376,2,0):
                support:    0.043478260869565216
                confidence: 0.06666666666666667
        >inf1009(A,2,B):
                support:    0.13043478260869565
                confidence: 0.19999999999999998
```

This result indicates that the about half of the students in the dataset fail a discipline in both the first and second semesters. The two most common disciplines to fail in the second semester in this situation are INF1008 and INF1009. Both are disciplines without any pre-requisites recommended for the second semester. Perhaps modifying the rules in the domain to require appropriate pre-requisites for these disciplines could positively impact the performance of students.

This example highlights the kinds of insights that the discipline of Process Mining aims to provide process administrators and managers. We further discuss concrete examples in the educational domain in the next chapter.

3.4.2.3 Additional interest metrics

We posit that with the mechanisms used to compute the support and confidence of rules described above could be used to compute other metrics of interest and relevance. In the literature of association rules, one such typical measure is the *lift,* used to determine the strength of the association rule. We will define the concept of lift for goal-inference rules and discuss other possible metrics.

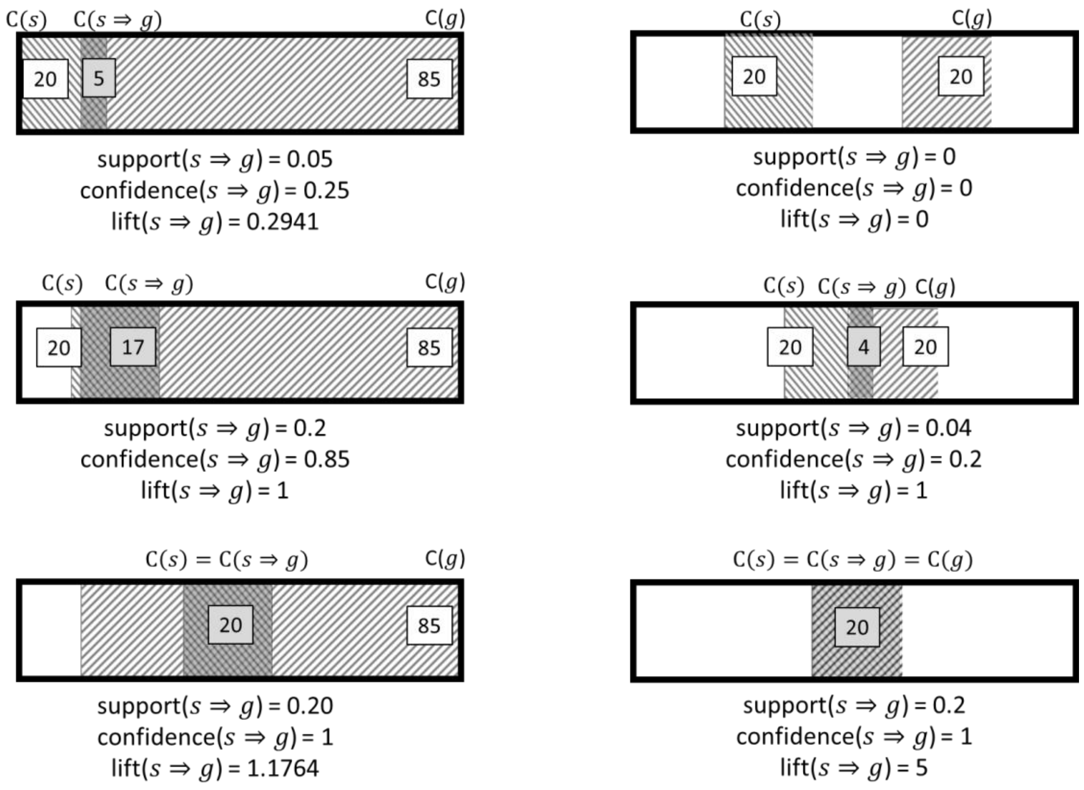
In association rule mining the lift is the ratio between the rule's confidence and the expected confidence, with the assumption that there is no statistic relation between the LHS and RHS. Formally, "the lift of a rule is defined as lift(X⇒Y) = supp(X∪Y)/(supp(X)supp(Y)) and can be interpreted as the deviation of the support of the whole rule from the support expected under independence given the supports of the LHS and the RHS". (HORNIK, GRÜN e HAHSLER, 2005). An alternative (and equivalent) way to compute the lift is lift(X⇒Y) = supp(Y)/confidence(X⇒Y)/

A lift value of 1 indicates that the antecedent and consequent are independent of each other; a value higher than 1 indicates that the antecedent has a positive effect on the consequent, and a value lower than one indicates a *negative* effect (that is, the lower the lift, the more we expect to *not* observe Y given X).

Intuitively, we reason: if X e Y were independent sets, we would expect a supp(X∪Y)= Q. If the actual support is greater than Q, then the lift will be greater than 1. If the actual support is lower than Q, the lift will be smaller than one.

In our approach we define the lift($s \Rightarrow g$) of the goal-inference rule as $\frac{\text{support}(s \Rightarrow g)}{(\text{support}(s)\text{support}(g))}$, where the support($g$) is defined similarly to the support(s). That is, with C(g) as the set of cases that reach the goal – but not necessarily the situation - and support(g)=$|\frac{\text{C}(g)}{C}|$. An alternative (and equivalent) way to compute it is lift($s \Rightarrow g$)= $\frac{\text{confidence}(s \Rightarrow g)}{\text{support}(g)}$.

The mechanism implemented in the Library of Typical Plans for Process Mining for computing the support of goal g is the same as for computing the support of situation s. Hence, we obtain an additional measure of how relevant the relation between the situation and goal is. We expect to find lift values greater than one for goals that *positively correlate with the situation* in the domain. A visual representation of this interpretation is given in Figure 3.



C(s) C($s \Rightarrow g$) C(g)
20 5 85
support($s \Rightarrow g$) = 0.05
confidence($s \Rightarrow g$) = 0.25
lift($s \Rightarrow g$) = 0.2941
C(s) C(g)
20 20
support($s \Rightarrow g$) = 0
confidence($s \Rightarrow g$) = 0
lift($s \Rightarrow g$) = 0
C(s) C($s \Rightarrow g$) C(g)
20 17 85
support($s \Rightarrow g$) = 0.2
confidence($s \Rightarrow g$) = 0.85
lift($s \Rightarrow g$) = 1
C(s) C($s \Rightarrow g$) C(g)
20 4 20
support($s \Rightarrow g$) = 0.04
confidence($s \Rightarrow g$) = 0.2
lift($s \Rightarrow g$) = 1
C(s) = C($s \Rightarrow g$) C(g)
20 85
support($s \Rightarrow g$) = 0.20
confidence($s \Rightarrow g$) = 1
lift($s \Rightarrow g$) = 1.1764
C(s) = C($s \Rightarrow g$) = C(g)
20
support($s \Rightarrow g$) = 0.2
confidence($s \Rightarrow g$) = 1
lift($s \Rightarrow g$) = 5

Figure 3 - A visual representation of the lift of goal inference rules.

In this Figure 3 each example is again a domain with 100 cases (the box), the set of cases reaching the situation C(s) is given by the left-leaning (\\\) hashed lines, and the set of cases reaching the goal C(g) is given by the right-leaning (///) hash lines. In all scenarios the support(s)=0.20, and we vary the values of the support(g). The examples on the left are the ones with a larger support(g)=0.85. The examples on the right are the ones with a smaller support(g)=0.20.

In the middle row we have lift($s \Rightarrow g$) = 1 for both cases. This means the support of the rule is exactly as we would expect in that scenario. Notice that when the support(g) is larger we require a proportionally larger support($s \Rightarrow g$) to obtain a lift of 1. It's easy to see that the same is true for the support(s) (both support(g) and support(s) are in the denominator of the computation).

In the bottom row we have the maximum lift achievable with the support(s) for both cases. The maximum lift is obtained when the confidence is maximum, support(s)=support($s \Rightarrow g$). On the left, we see that the maximum lift is 1.1764. On the right, the maximum lift 5. The latter case easily reflects the intuition: a support of 0.2 is 5 times greater than the expected support 0.04.

In the example above the goal of rule 8-1 – to graduate – is achieved by 5 out of the 23 students in the domain. Hence, the support(g) is 0.2173. Given that supp($s \Rightarrow g$)=0.1304 and that support(s) = 0.4782, we have a lift($s \Rightarrow g$) = 1.25. The value a little higher than 1 indicates that the goal is (weakly) positively correlated with the situation – this is unexpected, as the situation is a 'negative' situation. This insight is used in Section 3.5 to drive the generation of additional goal-inference rules. We will discuss this rule again in the model enhancement, when repairing the model to account for additional cases (see Section 3.5.3).

The literature describes many other metrics of interest for frequent itemsets, association rules and frequent sequences. We discuss the implementation of the mechanism for computing the ones discussed in (ZAKI e MEIRA JR., 2014, p. 301-309) in Section 4.2.2.

Consider the additional example below.

```
gi_rule( 3-3, student(Student),                                    %Agent
           ( student(Student), app(Student,inf1403,_),           %Situation
             app(Student,inf1005,_), app(Student,mat1200,_),
             app(Student,mat1161,_), app(Student,fis1033,_),
             app(Student,fis1034,_) ),
         grad(Student)                                           %Goal
).
```

This rule relates to students that eventually complete the first recommended semester, and how they proceed to graduate. The plans mined from this rule include the plan with a generic operation, and the respective cases, as follows:

```
complex : (generic : g2(S),
           frame : [student/o],
           specializations : [c19(S),c16(S),c9(_),c4(_)],
           cases : [2388,2387,2375,2371])
```

In this case:

```
Support of Rule 3-3:
    support:    0.17391304347826086
    confidence: 0.8
    lift:       3.6799999999999997
    leverage:   0.1266540642722117 % Additional interest metrics
    conviction: 3.9130434782608705 %
```

The support($s \Rightarrow g$) of the rule is 0.1739. The situation s in this rule is reached by 5 out of the 23 students in the domain – hence, support(s) = 0.2173. Thus, the confidence($s \Rightarrow g$)=0.8, and the lift($s \Rightarrow g$) =3.6799. This means that although the rule is *infrequent*, we have a high *confidence* that students that eventually complete the disciplines recommended for the first semester will graduate, and also find that reaching that situation (strongly) positively influences the goal of graduating.

3.5 Model Enhancement

The third task in Process Mining is the *model enhancement*. The goal of this task is to adapt the model to the needs for analysis and reasoning about the domain. It may comprise making the model more representative of reality, either by generalizing or extending the model. It may also comprise repairing the model to make it more accurate or precise

There are many possible model enhancement approaches. We select a few, based on the characteristics of our example educational domain, to showcase the flexibility and power of representation of the planning approach based on a conceptual model. A useful classification of enhancement tasks is that of *model repair* tasks (for a more faithful model) and *model extension* tasks (for a model that represents more kinds of behaviors). These are related, and there are intersections in goals and methods among them.

Typically, the *repair* approaches are motivated by issues identified in the conformance checking. The goal of the model repair tasks is to align process model with the actual events in the domain. In our example domain the conformance checking task detects the inability of the model to replay plots, as discussed in Section 3.4.1. We repair the model to account for the discrepancies between the plots and the model. These discrepancies can happen due to incomplete, missing, or erroneous information in the log – see the discussion on *log quality* in Section 2.1. Assuming that the information in the log is correct and that the plots (the traces) are sound, the discrepancies may indicate the model is *incorrect* and/or *incomplete*. In our approach, these issues are dealt with by amending the conceptual model.

The *model extension* approaches are motivated for the need of more general representation of behavior in the domain, and for the addition of *perspectives* to the model. Typically, these approaches motivated from domain-dependent analyses and, in the case of additional perspectives, are reliant on additional case and event attributes from the event log (VAN DER AALST, 2011).

3.5.1 Model repair

The task of model repair relates to the discrepant or deviant behavior between the event log and the process model, typically identified via the conformance checking task.

There are several possible explanations for deviations between the traces and the model. One of them is that the *logging* process is incorrect, e.g. with noise, missing or incorrect data. We will not consider that scenario, but it is discussed in the literature (LY, INDIONO, *et al.*, 2012; SURIADI, ANDREWS, *et al.*, 2017).

Another possibility is that the model is incomplete or describes events in the wrong level of detail, and thus is incapable of replaying the trace. In that scenario we are concerned with the representative power of the model. We discuss the generalization, simplification or extension of the model in the next sections.

Finally, the deviations may indicate that the cases are performing the process in ways that are outside of the boundaries determined by regulations or systems. These deviations between the event log and process model represent *forbidden* behavior that should not have been allowed. Capturing such behavior is the goal of the auditing task. Van der Aalst defines the *auditing* of process:

> “The term *auditing* refers to the evaluation of organizations and their processes. Audits are performed to ascertain the validity of and reliability of information about these organizations and associated processes. This is done *to check whether business processes are executed within certain boundaries set by managers, governments and other stakeholders.* For instance, specific rules may be enforced by law or company policies and the auditor should check whether these rules are followed or not. Violation of these rules may indicate fraud, malpractice, risks, and inefficiencies. Traditionally, auditors can only provide *reasonable assurance* that business processes are executed within the given set of boundaries. They check the operating effectiveness of controls that are designed to ensure reliable processing. When these controls are not in place, or otherwise not

functioning as expected, they typically only *only check samples of factual data*, often in the ‘paper world’”.

(VAN DER AALST, 2011, p. 193)

Thus, the auditing provides actionable insights for domain stakeholders to act. For example, detecting the deviations for ongoing traces can raise alarms. Alternatively, management may wish to visualize reports of discrepancies over time to pinpoint the reasons, culprits or resources responsible for the deviant behaviors. Finally, the process designers may wish to amend the model, “promoting” the deviant behavior to become the new normative pattern. In any case, these are domain-dependent decisions and will require domain knowledge input from specialists and stakeholders.

Finally, the discrepancies may occur due to the inability of the model to represent the actual behavior in the domain – that is, due to the model being *incorrect*. In this case, we fix the mode by making the appropriate changes.

In our approach, this will comprise identifying the operations that are most commonly discrepant in the cases’ plots, and amending their definitions, preconditions and effect accordingly.

Recall our example from Section 6.1 in which we identified that a number of disciplines in the student’s plots that are of other departments (such as PSI1849 and PSI1847 in our example) are discrepant. Assume that in the analysis of this cases we find that they are *exceptions* to the rule of pre-requisites of disciplines.

Hence, in this example our *repair* task consists of removing, for disciplines of other departments, the preconditions of disciplines from that department. For example, we would retract the clause:

```
precond( psi1849 (Student, SemesterA, _Grade),
         ( success(Student, psi1848, SemesterB), SemesterB < SemesterA).
```

A consequence of the way we model the operations based on *activity clauses* is that this kind of repair can be performed *a priori* by altering the activity clauses of these operations. Recall that we define the operations based on the set of activity clauses $\mathcal{A}$. We could remove each discipline d from the list of pre-requisites of each activity $a \in \mathcal{A}$ when $\mathcal{A}[a]$ and $\mathcal{A}[d]$ start with that department code. Re-generating the dynamic schema from this repaired set of activity clauses would fix the problem by construction.

Recall also from Section 3.4.1 that we identified that, for several cases, the requirements for the case termination operation `graduate` were not being met. This happened even when we ‘allowed’ all other discrepant operations in the replay.

Assume that we are able to verify that is due to a mismatch between the process model and the actual process. In the educational domain, the rules for graduation are much more flexible than a concrete and predefined set of disciplines. They involve *elective* disciplines organized in *groups*, such that the student must be successful in one (or several) disciplines of the *group* before graduating. This motivates the generalization of the model to capture this additional behavior. In the Section 3.5.2 we will demonstrate how this generalization (with respect to the groups of disciplines) can be performed. We will also discuss other kinds of model repair and generalization by modeling additional behavior.

Finally, consider the following cases from the replay of our cases:

```
%With the failed preconditions --suppressed

Case 2369
% Allowed > Failed Operations
[]  >  [dropout(2369,1)]

Case 2372
[inf1761(2372,6,57)]  >  [dropout(2372,8)]

Case 2373
% Allowed > Failed Operations
[]  >  [dropout(2373,4)]

Case 2379
% Allowed > Failed Operations
[inf1771(2379,8,50)]  >  [dropout(2379,10)]
```

This is a subset of the students that drop out without failing the same discipline several times (the precondition we determined for the `dropout` operation). As for the graduation operation, we verify that this is due to a mismatch between the process model and the actual process. In Section 3.3.4 we discussed the definition of the case termination operations’ preconditions as domain dependent and how they might relate to *exogenous events*. These are harder to capture than those based on failures and successes in activities and time limits. This is the case for the `dropout` operation.

Upon analysis of the domain, we find that students that abandon the program are indistinguishable from the ones that are terminated due to domain rules. This highlights the importance of having more granular and accurate information of the case terminations for the modeling. We will discuss the addition of new case termination operations in Section 3.5.3.

3.5.2 Model generalization and additional behavior

Recall that the tradeoff between *fitness*, *precision*, *generalization* and *simplicity* of the model. In this section we discuss incrementing the model to represent more complex activities, promoting the *generalization* of the model at the cost of the *simplicity* (the model becomes intrinsically more complex).

As we introduced in the previous section, one characteristic of the academic program domain that is not represented in the formulation used so far is that of *elective disciplines*. These are *groups* of disciplines from which the student must be approved in one or several of them in order to graduate. The graduation requirements involve a system for assigning *credits* to the student based on successful completion of disciplines, and the grouping of *elective* disciplines in *groups* dictates how many of those disciplines are required (but not a specific subset).

We'll deal with the modeling of groups of disciplines in an example. Assume that in the analyses of the typical plans the program administrators find that it is important to represent this behavior. For the purposes of exemplifying the extension of the conceptual model we will model a simplified mechanism of group disciplines, in which the student must choose a single discipline from the group (instead of possible several).

Recall our definition of `activity` clauses obtained from the domain. We additionally obtain, in a similar fashion, a set of `group` clauses of the format:

```
<group-clause> ::= group(<group-name>,[<group-activities>]).
<group-activities> ::= <activity-name> | <activity-name>,<group-activities>
```

For example, with several disciplines suppressed for the first two groups:

```
group(cre0700,[cre1112, cre1113, ..., teo1802]). %17 suppressed
group(fil0300,[fil1000, fil1002, ..., fil1814]). %21 suppressed
group(inf0310,[inf1624, inf1629]).
group(let0310,[let1011, let1113, let1910]).
group(inf0300,[inf1600, in1612]).
```

The group names are added to the set of activy name $\mathcal{A}$ and thus will figure as activities in the result and failure clauses. For example:

```
app(2368, inf0310,2) %Student 2368 completed discipline of group inf0300 in semester 2
```

The groups will not appear as operations in the student's plots. However, they may appear as part of situation and goals in goal-inference rules. Consider the following goal-inference rule:

```
gi_rule(12,
        student(Student),
        (   student(Student), app(Student,inf0310, _) ),
        dropped(Student) ).
```

This rule represents the students that complete the group INF0310 and later dropout. The approval in the group can be achieved via the approval in any discipline in the group, in this simple formulation. Hence, we define the approval in the group as an additional *effect* of the activities. We define additional `added` clauses to represent this. In the example, the approval in group INF03010 can be achieved via the approval in either discipline INF1624 or in INF1629. The mining of the plans with this rule yields a single plan, for student 2372:

```
simple: ( single: dropout(2372,8),
          cases:[2372],
          constraints:[] )
```

The rule has low support (it is infrequent in the domain); low confidence (it is uncommon among students that are approved in the group); and low lift (dropping out is inversely correlated with completing the group):

```
Support of Rule 12:
    support:    0.041666666666666664
    confidence: 0.16666666666666666
    lift:       0.26666666666666666
```

Although the mechanism is simple, it enables interesting additional results. For example, the following goal-inference rule:

```
gi_rule(13,
        student(Student),
        student(Student),
        ( app(Student,cre0700,_), app(Student,fil0300,_) )
        ).
```

This rule represents the students that eventually complete both groups CRE0700 and FIL0300 of *optative* disciplines (see the Appendix). The mining of typical plans using this rule yields the following plans:

```
complex : (generic : g1(A),
             frame : [student/o],
   specializations : [c4(A),c3(A),c2(_),c1(_)],
             cases : [2389,2388,2387,2375,2372,2371,2370,2368],
       constraints : []),

complex : (composite : c4(2389), (... )), % suppressed, 1 case

complex : (composite : c3(2371), (... )), % suppressed, 1 case

complex : (composite : c2(B),
               frame : [student/o],
          components : [ f1 : cre1100(B,2,C),
                         f2 : cre1127(B,D,E),
                         f3 : fil1000(B,F,G) ],
        dependencies : [f1-f2],
               cases : [2388,2375,2372,2370],
         constraints : [ (C in 91 .. 100), (D in 3 .. 6), (E in 90 .. 100),
                         (F in 6 .. 10),   (G in 50 .. 88) ] ),

complex : (composite : c1(H),
               frame : [student/o],
          components : [ f1 : cre1100(H,2,I),
                         f2 : cre1116(H,J,K),
                         f3 : fil1000(H,L,M) ],
         dependencies : [f1-f2],
               cases : [2387,2368],
         constraints : [ (I in 70 .. 87), (J in 3 .. 5), (K in 90 .. 100),
                         (L in 7 .. 9),  (M in 57 .. 95) ] )
```

Although there are several typical plans, one of them (`c2`) is much more common than the others. This is further highlighted by the analysis of the metrics of interest:

```
Support of Rule 13:
    support:    0.34782608695652173
    confidence: 0.34782608695652173
    lift:       1.0
        >g1(A):
                support:    0.34782608695652173
                confidence: 0.34782608695652173

       >c4(2389):
                support:    0.043478260869565216
                confidence: 0.043478260869565216
        >c3(2371):
                support:    0.043478260869565216
                confidence: 0.043478260869565216
        >c2(A):
                support:    0.17391304347826086
                confidence: 0.17391304347826086
        >c1(A):
                support:    0.08695652173913043
                confidence: 0.08695652173913043
```

This is an example of how the model repair task can be performed by adding functionalities to the domain.

Suppose that we found necessary to model the system of credits obtained by the students. We could capture that behavior in our the model by modifying the representation of discipline operations, their pre-requisites and their effects, and by leveraging *negative* effects (the deletion of facts from the state): the *effect* of successfully completing a discipline would comprise *deleting* a fact that states the current number of credits obtained by the student and *adding* a fact that states the new number of credits. For ease of representation, we won't deal with that scenario in our educational domain examples, but similar behavior is captured in the examples in a domain with similar characteristics in (GOTTIN, DE LIMA e FURTADO, 2015).

3.5.3 Model generalization and additional cases

Model generalization is especially important when considering that additional cases can be observed as the event log progresses. Recall that in our approach we consider *partial traces*. Hence, the event log may be periodically incremented. Some cases will have new operations appended to their plot; some cases will terminate; and some entirely new cases will be admitted in the process.

Here we will consider a different example – recall that our previous examples have dealt with a domain of 23 students. These are students that share a characteristic (the same academic term of first enrollment). We will now consider an additional set of 24 students (admitted, i.e., first enrolled, in a following academic term). Regardless, the modifications of the model that incur are exemplary of the kinds of the generalization tasks that are useful for the online admission of new cases.

In this example the set of cases $\mathcal{C}$ will be updated. We will refer to the original set of student as $\mathcal{C}^1$ and the new set of students as $\mathcal{C}^2$, so that $\mathcal{C} = \mathcal{C}^1 \cup \mathcal{C}^2$. Furthermore, we define a new case attribute $enrollment$ such that $\#_{enrollment}(c) = a$ for all $c \in \mathcal{C}^1$, and $\#_{enrollment}(c) = b$ for all $c \in \mathcal{C}^2$.

Recall our discussion from Section 3.3. We originally defined the `<entity-clause>` for the case entity as `entity(student)`, with no entity attributes. Recall, however, that the conceptual model allows for such attributes in its formulation. Even though we leveraged the $status$ case attribute for the definition of the case termination operations in Section 3.3.2.1.2 we didn't consider it for defining an attribute of the case entity since *not all cases have a status value* (some are still ongoing traces).

We have an $enrollment$ value for each student. Hence, with the additional cases, we will consider the $enrollment$ attribute to compose the `<attribute-list>` of the case entity clause. Instead of a unary predicate, the case entity will be extended to include one argument. In this case:

```
% The case entity with attribute
entity(student, enroll).
```

Where the atom `enroll` is given by the mapping $Q[enrollment]$.

The change in the definition of the case entity in the definition of the static schema will reflect in the assertions of the initial state of plots (see Section 3.3.5). The plots still refer to a single case each, so it suffices to define the initial state with the additional `enroll` attribute. The general formulation is still:

```
initial_database( <case-id>, <entity-clause> ).
```

But now the example of the initial state for the plot of student 2367 is:

```
initial_database( 2367, student(2367, a) ). % Student 2367 from the original set
```

Similarly, the definitions of the goal-inference rules must change accordingly with respect to the agent, situation and goal terms. Wherever the case entity is used in the rule we add the additional argument to the clause. For example, the goal-inference rule with id 3 introduced in Section 3.3.3:

```
% Also updated to use the app and rep functors for the success and failure terms
gi_rule( 3,
    student(Student, Enroll),                               % Agent
    ( student(Student, Enroll)                              % Situation
      not app(Student, _, _), not rep(Student,_, _) ),
    ( not failure(Student,_, _), app(Student, inf1403, 1),  %Goal
      app(Student,inf1005,1), app(Student,mat1200,1), app(Student,mat1161,1),
      app(Student,fis1033,1), app(Student,fis1034,1) ) ).
```

The log $\mathcal{L}$ is also extended to contain the traces $\hat{c}$ for all $c \in \mathcal{C}^2$. Recall TABLE from Section 3.2 representing the log with the traces of students $\mathcal{C}^1$. We show a sample of the traces of the appended log $\mathcal{C}$ below in TABLE:

Table 3 An excerpt from the extended event log

Event id	**Case id**	**Event Attributes**						**Case Attributes**	
		timestamp	**activity**	**credits**	**class**	**lecturer**	**grade**	**enroll**	**status**
77273	2368	8	INF1413	4	3WA	Jessica Leon	57	a	
77417	2370	7	INF1636	4	3WA	Crystal Landry	57	a	dropped
…	…	…	…	…	…	…	…	…	…
79665	2402	3	FIS1033	4	33V	Caitlin Andrews	5	b	halted
80879	2419	10	INF1951	2	3WE	Ryan Gallagher	100	b	graduated

Notice that we choose to represent the case attributes $enroll$ and $status$ explicitly in the Table. This is just a notation convenience – in any case, the Process Mining approach assumes, since a case can be uniquely identified for each event, that the case attributes can be associated to the event without issues.

The addition of these cases to the domain also introduces a new value for the $status$ attribute. This can be seen in the example of student 2402, who has *halted* the academic program. We additionally revise the entire dataset (including the original cases) and assign the $status$ attribute of the *transferred* students to *halted* as well.

In Section 3.3.2.1.2 we defined the case termination operations based on this attribute. In the original dataset, besides ongoing traces, we had only *graduated* and *dropped out* students. Now we also have *halted* students.

Thus, the extension of the model with the additional cases incurs in the definition of a new case termination operation, a `<case-termination>` term and the operations preconditions and effects. In this case, the operation does not have any preconditions.

```
operation( halt(Student, Semester), [student/o, semester/in] ).
% <case-termination> = halted/1
added( halt(Student, Semester), halted(Student) ).
```

The new case termination operation can be used as goal in goal-inference rules. An example is given below:

```
gi_rule(20, student(Student,Enroll),
           ( student(Student,Enroll),
             rep(Student,_D1, 1
           ),
           ( halted(Student),
             rep(Student,D,N1),
             rep(Student,D,N2),
             N2 #= N1+1 ) ).
```

This rule aims to capture the plans of students who started the course with a failing discipline, who later halted the course having at least two consecutive semesters with failed activities. The results of the mining of plans for this rule in the extended model are given below.

```
complex : (generic : g1(S),
           frame : [student/o],
           specializations : [c2(_),c1(_)],
           cases : [2417,2382] ),

complex : (composite : c2(2417),
           frame : [student/o],
           components : [ f1:inf1626(2417,9,0),
                          f2:inf1626(2417,10,26),
                          f3:halt(2417,10) ],
           dependencies : [],
           cases : [2417],
           constraints : []),

complex : (composite : c1(2382),
           frame : [student/o],
           components : [f1: adm1019(2382,11,16),
                         f2:adm1019(2382,12,39),
                         f3:halt(2382,12)],
           dependencies : [],
           cases : [2382],
           constraints : [])
```

This identifies disciplines INF1626 and ADM1019 as the disciplines the students failed repeatedly before leaving the program or transferring.

The results of the interest metrics are as follows.

```
Support of Rule 20:
    support:    0.0425531914893617
    confidence: 0.06451612903225806
    lift:       1.5161290322580645
```

Which shows this is infrequent behavior (*support* of 0.04) in the domain, but somewhat interesting (a *lift* of 1.5).

Another consequence of adding more cases is that the *support*, and therefore the other fitness metrics, of the rules and plans are changed.

For example, recall the rule 8-1 originally introduced in Section 3.4.2.2. and the results obtained for the interest metrics in the original cases. We discussed in Section 3.4.2.3 that the lift higher than 1 was an unexpected find, since it indicates a (weakly) positive correlation between a ‘bad’ situation (a semester without any approvals in disciplines) and a ‘good’ outcome (graduation):

```
Support of Rule 8-1: % Over the original set of cases C1
        support:    0.13043478260869565
        confidence: 0.2727272727272727
        lift:       1.2545454545454546 % <- unexpectedly high
```

After the addition of the cases in $\mathcal{C}^2$ and the repair discussed above, we mine the typical plans for this rule again. We obtain the following

```
Support of Rule 8-1: % Over the complete set of cases C = C1 U C2
    support:    0.06382978723404255
    confidence: 0.13636363636363635
    lift:       0.712121212121212 % expectedly lower
```

The support of the rule still indicates that the mined plan is very infrequent – still only one student performs that plan. With the additional cases considered, however, we have more *graduated* students – more students whose plots reach the rule’s goal – and the resulting *lift* is substantially lower, which is intuitively expected. This case illustrates how the evaluation of process models is susceptible to *infrequent behavior*. The generalization of the model will tend to include more frequent behavior, making the fitness metrics ‘converge’ towards well-behaved values. In the next section we discuss possible approaches for model simplification.

3.5.4 Model simplification

Recall again the tradeoff between *fitness*, *precision*, *generalization* and *simplicity* of the model. Suppose that, contrary to the scenario in Section 3.5.3 we find that the process model is representing behavior in *too much* detail. Perhaps too much *infrequent behavior*, a well-known issue in the literature (LEEMANS, FAHLAND e VAN DER AALST, 2013; MANHARDT, DE LEONI, *et al.*, 2009), dominates the analyses. This motivates the simplification of the model, promoting the *simplicity* and, in some cases, the *precision* (the model will generate less infrequent behavior that is not present in the event log) at the cost of the *generalization* and, in some cases, the *fitness* of the model.

In graphical models, the simplification of the mode typically incurs in removing paths that are not taken by a representative amount of cases. In our approach, the simplification of the model can be achieved via filtering of the infrequent or irrelevant behavior. For example, we may remove infrequent behavior by removing from the model the operations that are used sparsely (or not at all) used by students and don’t figure in typical plans. This implies filtering the *traces* of the cases and simplifying the model itself.

In our domain, this filtering impacts on the removal of the *operation clauses* corresponding to these activities from the students’ plots. Furthermore, it leads to simplifying the dynamic schema by retracting the *operation definition clauses*. In the latter case, this includes the retraction of the effects and preconditions of the operations as well. Finally, the result clauses referring to these operations in the preconditions and effects of all other operations must also be removed.

For example, consider operation INF1006. It does not figure in any of the plots and typical plans, hence it is infrequent and can be filtered out. It has three sets of alternative preconditions, so we retract all the following the clauses:

```
operation( inf1006(Student,Timestamp,Grade),
           [student/o, semester/in, grade/with] ).

precond( inf1006(Student, SemesterA, _Grade),
        ( success(Student, inf1001, SemesterB), SemesterB < SemesterA).
```

```
precond( inf1006(Student, SemesterA, _Grade),
        ( success(Student, inf1004, SemesterB), SemesterB < SemesterA).

precond( inf1006(Student, SemesterA, _Grade),
        ( success(Student, inf1381, SemesterB), SemesterB < SemesterA).


added( inf1006(Student, Semester, Grade), app(Student,inf1006,Semester)):-
      passing_grade(Grade).
added( inf1006(Student, Semester, Grade), rep(Student,inf1006,Semester)).
```

Finally, the discipline is a pre-requisite for several other disciplines, such as INF1018 which depends on either INF1006 or INF1007. Hence, we remove the clause:

```
precond( inf1018(Student, SemesterA, _Grade),
        ( success(Student, inf1006, SemesterB), SemesterB < SemesterA).
```

It easy to see that the filtering of operations will allow for *more* behavior in the domain, hence *generalizing* the model, given that it removes pre-conditions of operations.

Earlier we discussed the repair of the set of activity clauses to remove certain pre-requisites from certain disciplines, and how the re-generation of the dynamic schema from that repaired model would reflect the changes. A similar reasoning applies here. This simplification of the dynamic schema can be performed *a priori by removing the activity clauses* of infrequent operations. Since we define the operations based on the set of activity clauses $\mathcal{A}$, a process for filtering the atoms $\mathcal{A}[x]$ for every infrequent activity x, and re-generating the dynamic schema would reflect the changes in the entire model by construction.

3.5.5 Model extension and the generation of Goal-Inference Rules

We discussed in Section 3.3.3 how the goal-inference rules originally defined represent normative patterns in the domain. The rules represent *expected* behavior, and the mining of typical plans based on the rules shows *how* the cases conform to the pattern. In Section 3.3.4 we discussed how the definition of this rules depends on domain knowledge and how the (semi-)automatic discovery could be leveraged for the process discovery. In (FURTADO e CIARLINI, 2001) the task of discovering rules is anticipated as a difficult knowledge-discovery task.

In our approach we will perform this discovery by manipulating the conformant patterns, typically by relaxing, splitting and combining the rule(s) situation(s) and goal(s). This will take into account the fitness metrics computed for the rules, as well as other domain-dependent insights.

Recall that goal-inference rules are defined with the agent a, situation s and goal g. Since we refer to rules about agents in the domain, we typically define the rules with the case entity as the agent. The situation and goal are defined as a term or conjunction of terms that must hold in the plot. Each term $s_1, s_2, \dots, s_n$ in the conjunction that is s can be totally or partially *lifted* – that is, with variables as arguments. The same holds for each of the terms $g_1, g_2, \dots, g_m$. These terms typically refer to the effects of operations, either success or failure terms for the activity operations or the termination effects given by case termination operations. In the success or failure terms, the first argument is typically a variable matching the variable in the first argument of the case entity.

In our formulation, the relaxation of a goal-inference rule may comprise *lifting* arguments of one or more terms with grounded arguments in the situation or goal; or *removing* one or more terms from the situation or goal.

The variables introduced in the terms by *lifting* can be co-designated. Since we defined the arguments of the success and failure clauses based on the event attributes that identify events (that are in the classifier, like the timestamp in our example), we may use that information for the co-designation of variables

The removal of terms from the conjunctions is straightforward. We choose one or more terms from the situation or goal of a goal-inference rule and delete them, generating a new rule. There will be many possible combinations of removal, and thus, to avoid an exponential number of new goal-inference rules we propose that the removing of terms from the situation and/or goal may be subject to preference assertions that limit that number of combinations.

One possible assertion is to not remove terms with co-designated variables in the situation or goal. Another one is not removing the terms referring to the results of case termination operations. Notice that removing constraints over the variables does not guarantee that the rule will be more general, hence, we do not consider constraint terms for removal.

With these assertions, the number of possible combinations is reduced but still possibly impractical. It is important to apply restrictions and to guide the generation of rules by lifting and removal of terms such that a reasonable, small yet representative, set of rules is generated. We discuss the implementation of some mechanisms above in our Library of Typical Plans for Process Mining in Section 4.3.

A third type of automatic goal-inference rule composition relates to the plots of interesting students. When we identify an interesting case of a student, we can use the state reached by that student's plot as a hypothetical situation for new goal-inference rules. For example, if we determine a rule a, s, g where s is the situation resulting from student 2368's plot and the goal g is `grad(Student)`, the process of plan-mining with that rule will capture the plans used by other students, when they were in 2368's situation, for graduation.

So far, we've only discussed the general mechanisms. We'll now show examples in our educational domain. In Section 4.3 we describe the implementation of some of the algorithms discussed here.

A first reasoning is that *infrequent* rules are not capturing the behavior of many cases. If the pattern is normative – such as the recommended order of disciplines, in our domain – we'd expect to find reasonable support for rules (depending, of course, on the analysis). For example, recall rule with `<rule-id>` 3 as originally discussed in Section 3.4.2.1. We find that only three students perform a plan following this rule in the original set of cases, and we obtained the following metrics for the plans mined with this rule:

```
Support of Rule 3:
    support:    0.13043478260869565
    confidence: 0.13043478260869565
    lift:       1.0
```

Assume that we, as domain stakeholders, find that the rule is too infrequent. We hypothesize that by relaxing the requirements for the application of the rule we would find more students that follow potentially *similar* plans. One way to perform this relaxation is to *lift* the situation and/or goal clauses – or parts of the conjunctions that comprise them. We generate a rule 3-1:

```
% Original rule – normative pattern –
% Student approved in all 1st-semester-disciplines, without any failures, in the
recommended first semester
gi_rule( 3, student(Student),
       ( student(Student), not app(Student, _, _), not rep(Student,_, _) ),
       ( not rep(Student,_, _), app(Student, inf1403, 1),
       app(Student, inf1005, 1), app(Student, mat1200, 1),
       app(Student, mat1161, 1), app(Student, fis1033, 1),
       app(Student, fis1034, 1)     ) ).

% Derived rule –
% Student _eventually_ approved in 1st-semester-disciplines, with potential delays but
% still no failed disciplines

gi_rule( 3-1, student(Student),
       ( student(Student), not app(Student, _, _), not rep(Student,_, _) ),
       ( not rep(Student,_, _), app(Student, inf1403, _),
       app(Student, inf1005, _), app(Student, mat1200, _),
       app(Student, mat1161, _), app(Student, fis1033, _),
       app(Student, fis1034, _)     ) ).
```

Rule 3-1 will capture the plans of students, from their original enrollment, until they are approved in the disciplines recommend for the first semesters – not necessarily in the first semester.

Notice that we lifted all of the timestamp variables in the rule at once. We could have chosen to lift others parts of the goal – perhaps only the semesters of disciplines from the mathematics department (MAT1161 and MAT1200), and still requiring approval in the student's first semester for the others. We also lifted the terms in the goal *independently* – the singleton variables for the semesters may all assume distinct values. We could have lifted a subset of the success clauses semester argument with the *same variable*, forcing the concomitant approval of disciplines. There are many combinations of these possibilities – which highlights the need for domain input, since exhaustively exploring the relaxation of rules would generate a huge number of combinations and comprise a significant computational cost.

In the case of this example, the relaxation of the semesters in the goal clauses still does not improve the support:

```
Support of Rule 3-1:
    support:    0.13043478260869565
    confidence: 0.13043478260869565
    lift:       1.0
```

This is unexpected - but can be verified in the original data – in the set of students C^1 *only* the students that are approved in the 1st-semester-disciplines *in the first semester* are eventually approved in all of them without failing a discipline. In other words, all other students either don't ever successfully complete the 1st-semester-disciplines, or fail at least one discipline before doing so.

Hence, the support of the rule is still the same – and low. We continue to relax the rule, this time by removing clauses from the situation and goal conjunctions. In our current example, the situation is already general – it represents the initial enrollment, reached by all students. Hence, we continue to relax the goal of rule 3-1 by removing terms from the goal. We illustrate below the goal 3-2, resulting from removing the requirement of not failing a discipline:

```
% Student eventually completes the 1st-semester-disciplines
gi_rule( 3-2, student(Student),
       ( student(Student), not app(Student, _, _), not rep(Student,_, _) ),
       ( app(Student, inf1403, _), app(Student, inf1005, _),
         app(Student, mat1200, _), app(Student, mat1161, _),
         app(Student, fis1033, _), app(Student, fis1034, _)      ) ).
```

The plans mined from this rule show that indeed there are more students that follow plans under these conditions:

```
complex : (composite : c1(_),
          frame : [student/o],
          components : [ f1 : fis1033(H,I,J), f2 : fis1034(H,1,K),
                         f3 : inf1005(H,1,L), f4 : mat1161(H,1,M),
                         f5 : mat1200(H,N,O), f6 : inf1403(H,P,Q)   ],
          dependencies : [],
          cases : [2388,2387,2375,2371,2368],
          constraints : [  (I in 1 .. 3),    (J in 54 .. 90),
                           (K in 81 .. 100), (L in 81 .. 99),
                           (M in 64 .. 89),  (N in 1 .. 2),
                           (O in 57 .. 72),  (P in 1 .. 5),
                           (Q in 69 .. 85)   ] )
```

The constraints over the semesters in the mined plan show that students typically are approved in the first semester in disciplines FIS1034, INF1005 and MAT1161, and delay the others. The metrics for this rule are:

```
Support of Rule 3-2:
    support:    0.21739130434782608
    confidence: 0.21739130434782608
    lift:       1.0
```

The support 0.2173 indicates that a little over a fifth of students *ever* obtain approval in the 1st-semester-disciplines. Consider another example of relaxation via the removal of clauses. In Section 3.4.1 we discussed how the rule with `<rule-id>` 2 was too restrictive, and that we found no plans of students to perform the 'perfect' recommended order of disciplines. Consider the simpler rule below:

```
gi_rule(10,
        student(Student),
        ( student(Student), not app(Student, _, _) ),
        (   not rep(Student,_,_), grad(Student)    ) ).
```

This rule will capture the plans of just-enrolled students until graduation without any failed disciplines. We find, for the original set of students:

```
Support of Rule 10:
    support:    0.043478260869565216
    confidence: 0.043478260869565216
    lift:       1.0
```

This rule is almost as restrictive as the one that implies the 'perfect' order of recommended disciplines – only 1 out of the 23 students performs a plan that satisfies these requirements. Hence, we're motivated to relax the rule:

```
gi_rule(10-1,
        student(Student),
        ( student(Student), not app(Student, _, _) ),
        grad(Student) ).
```

Now, the rule will capture the plans of all students that graduate. We obtain a higher support for the rule:

```
Support of Rule 11:
    support:    0.21739130434782608
    confidence: 0.21739130434782608
    lift:       1.0
```

While still a fraction of the students, the similar support of the rules 3-2 and 11 motivate us to investigate the relation between them further. Since the rule 3-2 represents students that *eventually* obtain approval in 1st-semester-disciplines, we indeed expect *all* of the students that perform a plan for rule 11 to perform a pan for 3-2.

To investigate that, we define a rule by combining rules 3-2 and rule 11. That rule is rule 3-3, already used in an example in Section 3.4.2.3:

```
gi_rule( 3-3, student(Student),                                           %Agent
           ( student(Student), app(Student,inf1403,_),                   %Situation
             app(Student,inf1005,_), app(Student,mat1200,_),
             app(Student,mat1161,_), app(Student,fis1033,_),
             app(Student,fis1034,_) ),
         grad(Student)                                                   %Goal
).
```

By combining the goal of rule 3-2 (as the situation) and the goal of rule 11 , this rule relates to students that eventually complete the first recommended semester, and how they proceed to graduate. Recall also from the previous example that we obtain the following metrics for this rule:

```
Support of Rule 3-3:
    support:    0.17391304347826086
    confidence: 0.8
    lift:       3.6799999999999997
```

As expected, the rule has a high confidence – and a high lift. However, we can identify that the *support is lower than that of rules 3-2 and 11*. By investigating the plans mined with this rule, and comparing the cases that perform plans for the three rules:

```
Cases of Rule 3-2
 [      2388, 2387, 2375, 2371, 2368]

Cases of Rule 10
 [2389, 2388, 2387, 2375, 2371]

Cases of Rule 3-3
 [      2388, 2387, 2375, 2371]
```

we identify that student 2389 performed a plan for rule 10 but not for rule 3-2, which is not normal in the domain. We explore the student 2389 and verify, via the replay of plots, that his plot is particularly problematic:

```
%With the failed preconditions --suppressed

Case 2389
% Allowed > Failed Operations
[]
>
[ mat1154(2389,-6,55), inf1631(2389,2,67),  inf1721(2389,4,13),
  inf1721(2389,6,52),  inf1405(2389,7,85),  inf1015(2389,8,85),
  inf1413(2389,9,70),  inf1640(2389,10,11), inf1640(2389,11,60),
  graduate(2389,12) ]
% (...)
% Allowed > Failed Operations
[ mat1154(2389,-6,55), inf1631(2389,2,67),  inf1405(2389,7,85),
  inf1413(2389,9,70),  inf1640(2389,10,11), inf1640(2389,11,60) ]
>
[ graduate(2389,12) ]
```

While some of the discrepant disciplines in the original plot are satisfied by allowing others (i.e. INF1721, which the student attempted twice, and INF1015), we find that even by allowing all of the other discrepant operations the student still should not have been able to graduate. Notice also that the student's plot contains a discipline with a *negative* timestamp. Upon investigation of the plot and data sources we identify this student as a *re-enrolled* student. The identification that this student is very particular in behavior could further motivate the simplification of the model by removing his plot from the domain, for example.

Regardless, in the example of the cases following plans for rules 3-2, 10 and 3-3, we also identify that 2368 is the only student that eventually completes the 1st-semester-disciplines that *does not graduate*.

We verify the plot and observe the reason is *the student is currently enrolled in the program*. Hence, we are able to use the fact that he has performed a plan for rule 3-2 as a strong indication that he will eventually graduate. We may also suggest to her the plans mined with rule 3-3, evidencing the possibilities of the application of our approach for recommendation purposes as part of the *recommendation* task of *online process mining* (VAN DER AALST, 2011, p. 256-257).

This motivates us to define rules based on the situations of the students. We select student 2368 as an interesting case – our only student that is not a terminated case seems to be poised for graduation. We compose a situation s^c that is comprised of the approvals in disciplines shared by 2368 and the students that graduate. We define rule:

```
gi_rule(3-17, student(Student),
               ( student(Student),  app(Student, inf1007, _),app(Student,
inf1403, _),app(Student, inf1951, _),app(Student, inf1377, _),app(Student,
inf1010, _),app(Student, inf1950, _),app(Student, cre1100, _),app(Student,
inf1636, _),app(Student, inf1013, _),app(Student, inf1014, _),app(Student,
inf1771, _),app(Student, inf1721, _),app(Student, cre1141, _),app(Student,
inf1626, _),app(Student, mat1162, _),app(Student, inf1640, _),app(Student,
inf1413, _),app(Student, inf1008, _),app(Student, inf1301, _),app(Student,
fis1033, _),app(Student, mat1161, _),app(Student, inf1009, _),app(Student,
inf1015, _),app(Student, inf1012, _),app(Student, eng1029, _),app(Student,
inf1631, _),app(Student, inf1383, _),app(Student, inf1011, _),app(Student,
let1113, _),app(Student, inf1016, _),app(Student, mat1200, _),app(Student,
inf1005, _),app(Student, inf1018, _),app(Student, fis1034, _),app(Student,
cre1172, _),app(Student, mat1154, _),app(Student, inf1019, _)  ),
              grad(Student) ).
```

And mine the plans from the other students:

```
plans : [ simple : (single : graduate(_,_) ',' cases : [2388,2387,2375,2371] ','
constraints : [])
```

These plans should configure a *suggestion* of the disciplines to perform for graduation – however, from 2368's situation, the students that graduated *did not perform any more disciplines*, as evidenced by the plan being composed of just the case termination operation `graduate`. Hence, we hypothesize that student 2368 has not yet graduated due to exogenous conditions not captured by our model.

In this example, the plan we extracted was not helpful. Also, we defined a goal inference rule that considers all students as agents – this is because, so far, we haven't explored the characteristics of agents. We will explore the addition of characteristics of cases for the generation of goal-inference rules in the next section.

We consider another case in our original dataset. Consider students 2370 and 2382 – these are two students that performed disciplines for many semesters but eventually dropped out. Suppose that they had *not* dropped out of the program. We apply the same reasoning as above under that hypothetical scenario. We compose a situation s^c that is comprised of the approvals in disciplines shared by them and the students that graduate. We define the rule:

```
gi_rule(3-36, student(Student),
              ( student(Student),  app(Student, inf1009, _),app(Student,
inf1005, _),app(Student, inf1804, _),app(Student, fil1000, _),app(Student,
inf1007, _),app(Student, inf1018, _),app(Student, inf1012, _),app(Student,
fis1034, _),app(Student, cre1100, _),app(Student, inf1010, _),app(Student,
inf1383, _),app(Student, inf1631, _),app(Student, let1113, _),app(Student,
inf1008, _),app(Student, inf1301, _),app(Student, inf1636, _) ),
              grad(Student) ).
```

And mine the typical plans from the domain:

```
complex : (composite : c3(2388), frame : [student/o],
           components : [ f1 : cre1172(2388,8,85),  f2 : inf1015(2388,8,80),
   f3 : inf1016(2388,8,87),  f4 : inf1920(2388,8,100), f5 : inf1950(2388,8,100),
   f6 : inf1951(2388,9,100), f7 : graduate(2388,10) ],
           dependencies : [f1-f7,f2-f7,f3-f7,f4-f7,f5-f7,f6-f7],
           cases : [2388],
           constraints : []),

complex : (generic : g1(A), frame : [student/o],
           specializations : [c3(A),c2(_),c1(_)], cases : [2388,2387,2375] ),

complex : (composite : c2(2387), frame : [student/o],
          components : [ f1 : cre1172(2387,9,92), f2 : inf1013(2387,9,80),
  f3 : inf1608(2387,9,57), f4 : inf1715(2387,10,50), f5 : inf1920(2387,10,100),
  f6 : inf1950(2387,10,90),f7 : inf1951(2387,14,75), f8 : graduate(2387,14) ],
          dependencies : [f1-f8,f2-f8,f3-f8,f4-f8,f5-f8,f6-f8,f7-f8],
          cases : [2387],
          constraints : [] ),

complex : (composite : c1(2375), frame : [student/o],
          components : [ f1 : inf1013(2375,10,69), f2 : inf1014(2375,10,98),
  f3 : inf1951(2375,10,80),f4 : graduate(2375,10) ],
        dependencies : [f1-f4,f2-f4,f3-f4],
               cases : [2375],
         constraints : [] )
```

With interest metrics:

```
Support of Rule 3-36:
    support:    0.13043478260869565
    confidence: 0.5
    lift:       2.3000000000000003
    leverage:   0.07372400756143667
    conviction: 1.565217391304348
```

In this case, we obtain more interesting plans. Although each one was performed by only one student, we can interpret each one as a *suggestion* for the (hypothetically non-dropped out) students 2370 and 2382. We can observe, for example, that the plan performed by student 2375 is *much shorter* than the other ones – it comprises only three additional disciplines. If *time to graduation* is a criteria, we could choose to suggest that plan instead of the others. We discuss the implementation of the mechanisms for generating goal-inference rules, with simple examples, in Section 4.3.1.

3.5.6 Model extension and Decision Mining

The model extension relates to adding *perspectives* to the model. This is done by cross-correlating additional information in the event log to the process model.

The general process mining approach is mostly concerned with the ordering of activities. In graphical models that is the “control-flow” perspective. Another perspective discussed in the literature is the *time perspective*, concerned with “the timing and frequency of events” (VAN DER AALST, 2011, p. 11). The modeling of activities with timestamps, accounting for intertask and multiple dependencies, achieves some of the goals of this perspective.

In this section we will discuss the addition of the *case perspective* to the model in our approach, and how it can be accomplished by enriching the conceptual model. The case perspective focuses on properties of the cases, to understand decision-making and analyze differences among them (VAN DER AALST, 2011, p. 215). Intuitively, the case perspective is added to the process model by using information of additional *case attributes*.

In particular, we will discuss the *decision mining* approach that aims to provide insights on the activities performed with respect to characteristics of the cases. As an example, (VAN DER AALST, 2011, p. 234-235) shows how an external classification model (a decision tree) can be used to provide insights on how to model certain behaviors in a graphical model.

Recall our example of an additional case attribute $enroll$ for our example domain. We will explore the differences between students from one enrollment and another. We’ll explore the *decision mining* to find the influence of the $enroll$ attribute in the execution of activities. In our domain, the plans are intrinsically a record of the behavior of students. Hence, we will not use external models, and will instead reason about the characteristics of the students that follow each plan. We will, for example, identify plans that are uniquely or mostly performed by students from a single enrollment.

Recall the extended model, with additional cases, from the example in Section 7.3. We will consider two additional sets of cases C^3 with 5 cases and C^4 with 4 cases , so that our complete set $C = C^1 \cup C^2 \cup C^3 \cup C^4$ for the following examples comprises 56 total cases. Like before, the additional sets of cases define the $enrollment$ of the students in the set - $\#_{enrollment}(c) = c$ for all $c \in C^3$, and $\#_{enrollment}(c) = d$ for all $c \in C^4$.

We’ll discuss the general method here and provide details on the formulation in Section 4.3.

We define a set Ψ of attributes that will be used for decision mining. In our domain, since we only have the $enrollment$ attribute in the set $\Psi = \{enrollment\}$.

Each relevant combination of attribute values in Ψ determines a *characteristic class*. In our example, since the $enrollment$ attribute has four distinct values `a`, `b`, `c` and `d`, we have four distinct characteristic classes in $\mathbb{W}$. We abuse the notation and name classes A, B, C and D, respectively.

Each characteristic class implicitly defines a unique set of cases by the values of the attributes of those cases. We call them A-cases, B-cases, C-cases and D-cases in our example. A-cases are the cases of our original set of students $\mathcal{C}^1$, for which the $enrollment$ attribute value is `a`.

Our first approach for the decision mining is to compute metrics of interest for each characteristic class performing plans of a rule.

Consider for example rule 9, that captures the behavior of students that fail in a discipline in the first and second semesters:

```
gi_rule( 9, student(Student,Enroll),
           ( student(Student,Enroll), rep(Student, _D1, 1) ),
           rep(Student, _D2, 2) ).
```

The plan mining yields several plans, among which the generic operation plan:

```
complex: (generic: g1(S), frame: [student/o],
    specializations: [ fis1034(S,_,_), geo1116(S,_,_), inf1403(S,_,_),
                       cre1100(S,_,_), inf1007(S,_,_), dsg1421(S,_,_),
                       mat1162(S,_,_), mat1157(S,_,_), c1(S),
                       nf1008(S,_,_), fis1033(S,_,_), inf1009(S,_,_) ],
    cases: [ 2421,2395,2418,2417,2416,2414,2412,2411,2407,2405,2403,2397,
             2390,2385,2384,2383,2382,2381,2379,2377,2376,2370,2368 ]
```

And the interest metrics of the rule and the plans:

```
Support of Rule 9:
    support:    0.4107142857142857
    confidence: 0.6571428571428571
    lift:       1.1870967741935483
    lift2:      1.187096774193548
    leverage:   0.0647321428571428
    conviction: 1.3020833333333333
       % Support of each plan (confidence --suppressed)
       % In descending order of   %Support
        >g1(2384):              0.41071
        >inf1008(A,2,B):       0.07142
        >mat1157(A,2,B):       0.07142

        >inf1009(A,2,B):       0.05357

        >fis1033(A,2,B):       0.03571
        >inf1007(A,2,B):       0.03571
        >c1(A):                0.03571

        >fis1034(2421,2,44):   0.01785
        >geo1116(2395,2,19):   0.01785
        >inf1403(2418,2,19):   0.01785
        >cre1100(2414,2,15):   0.01785
        >dsg1421(2403,2,0):    0.01785
        >mat1162(2397,2,21):   0.01785
```

The information above gives us insight on how representative the plans of rule 9 are in the domain. We can see that some operations are more frequent than others by the comparative plan support. It does not consider the characteristics of the cases that performed the plans, however.

To that end, we will compute the number of cases of each $enrollment$ that perform the plans shown above.

For the case of rule 9 we have:

```
Characteristic of Cases in Rule 9:
  Characteristic enroll_a: 11 cases
  Characteristic enroll_b: 10 cases
  Characteristic enroll_c: 1 cases
  Characteristic enroll_d: 1 cases
```

With this information at hand, we are able to compute metrics that can be of interest for the analysis of the case perspective.

First, we compute the relevance of the characteristic for performing a plan of this rule. We call this the *in-rule support*. The *in-rule support* for a characteristic class `is` given by the number of cases in that class that perform a plan in the rule divided by the number of cases that perform a plan in the rule. For example, the in-rule support for enrollment `a` is 11 (the number of cases of enrollment a that perform a plan in the rule) divided by 23 (the number of cases that perform a plan in the rule):

```
  enroll_a in-rule      : 0.4783
  enroll_b in-rule      : 0.4348
  enroll_c in-rule      : 0.0435
  enroll_d in-rule      : 0.0435
```

The example above shows that the *in-rule* support represents the proportion of the cases that follow the rule that have each $enrollment$. Notice that characteristic classes A and B are dominant – each responsible for little less than half the cases that follow plans of this rule. At first sight, we might discard this rule (when suggesting plans, for example) for students $enrollment$s `c` and `d`.

However, the number of students in each characteristic class are very different. In fact, classes A and B are much more frequent in the domain – 23 and 24 students, respectively, in contrast to 3 and 4 students of class C and D.

In order to take with this class imbalance into account we define more metrics. We also compute the relevance of the goal inference rule for the characteristic class. We call this the in-characteristic support, or *in-char* for short. That is given by the number of cases with the characteristic that perform the rule divided by the total number of cases with that characteristic. For example, the in-char support of enrollment `b` is 10 (the number of cases with that characteristic performing a plan in rule 9) divided by the number of total students with that characteristic (24).

```
enroll_a in-char      : 0.4783
enroll_b in-char      : 0.4167
enroll_c in-char      : 0.2000
enroll_d in-char      : 0.250
```

The results above mean that although C-cases and D-cases do not account for many of the cases that follow plans of rule 9, they are significantly representative of the behavior of the characteristic classes: 20% and 25% of the C-cases and D-cases, respectively, perform a plan of this rule. Notice that the *in-rule* and *in-char* support of A-cases is the same – by coincidence, there are 23 A-cases and there are also 23 cases in total following plans of rule 9.

As in the computation of the interest metrics for the rules, we find that we can also compute the metrics for the individual plans of the rule.

Below we see the metrics for the plans mined from rule 9 and the B-cases:

```
Characteristic of Cases in Rule 9:
  Characteristic enroll_b: 10 cases

|                              in-plan   in-char
inf1403(2418,2,19)             0.0435    0.0417
cre1100(2414,2,15)             0.0435    0.0417
inf1007(A,2,B)                 0.0870    0.0833
dsg1421(2403,2,0)              0.0435    0.0417
mat1162(2397,2,21)             0.0435    0.0417
```

```
mat1157(A,2,B)                  0.1304    0.1250
fis1033(A,2,B)                  0.0435    0.0417
```

Contrasting these results with the interest metrics obtained for these plans, shown above, we can see that the one of the most frequent plans (performing discipline INF1008) is not performed by any of the B-cases. Conversely, the other frequent plan (MAT1157) is even more relevant when we account for the *enrollment* characteristic of these cases.

The analysis so far gives us an idea of the frequency and relevance of the characteristic classes for the rules and plans. The in-rule (or in-plan) and in-char metrics show how representative the rule (or the plan) and the characteristic classes are to each other.

As in the discussion about interest metrics of plans, there are several other types of metrics that could be computed. As an example, assume that it would also be interesting to assess the similarity of W-cases and rule cases. To that end we could rely on a measure like the Jaccard set similarity. A full representation of the metrics for the current example is given below.

```
Characteristic of Cases in Rule 9: |
  Characteristic enroll_a: 11 cases
    in-rule      : 0.4783
    in-char      : 0.4783
    jaccard      : 0.3143
|                              in-plan  in-char
mat1157(A,2,B)                 0.0435    0.0435
c1(A)                          0.0870    0.0870
inf1008(A,2,B)                 0.1739    0.1739
fis1033(A,2,B)                 0.0435    0.0435
inf1009(A,2,B)                 0.1304    0.1304

  Characteristic enroll_b: 10 cases
    in-rule      : 0.4348
    in-char      : 0.4167
    jaccard      : 0.2703
|                              in-plan  in-char
inf1403(2418,2,19)             0.0435    0.0417
cre1100(2414,2,15)             0.0435    0.0417
inf1007(A,2,B)                 0.0870    0.0833
dsg1421(2403,2,0)              0.0435    0.0417
mat1162(2397,2,21)             0.0435    0.0417
mat1157(A,2,B)                 0.1304    0.1250
fis1033(A,2,B)                 0.0435    0.0417

  Characteristic enroll_c: 1 cases
    in-rule      : 0.0435
    in-char      : 0.2000
    jaccard      : 0.0370
|                              in-plan  in-char
geo1116(2395,2,19)             0.0435    0.2000

  Characteristic enroll_d: 1 cases
```

```
    in-rule      : 0.0435
    in-char      : 0.2500
    jaccard      : 0.0385
|                               in-plan  in-char
fis1034(2421,2,44)              0.0435    0.2500
```

Notice that this analysis may also be used to fundament the suggestion of plans for cases based on their characteristic class. We define a basic suggestion mechanism as follows. For a case c, we obtain the characteristic class W by the checking the values of the attribute $\mathbb{W}$ in c. Given a rule r, we suggest the plans with the *highest* or *lowest* in-char support, depending on the scenario.

Suppose that we identify a new student 2500, a B-case – that is, a student with characteristic class $enrollment_{\text{b}}$ – that has failed in a student in his first semester. Hence, his situation applies to the situation of goal 9. Suppose that at the time in which he has to choose the disciplines for the 2nd semester he has access to the recommendation tool:

```
?- case_characteristic_similarity(2500, 9).
```

To obtain the recommendations, via backtracking:

```
Ratio :0.43478260869565216
Suggestion of plan in Rule: inf1403(2418,2,19);
Suggestion of plan in Rule: cre1100(2414,2,15);
Suggestion of plan in Rule: dsg1421(2403,2,0);
Suggestion of plan in Rule: mat1162(2397,2,21);
Suggestion of plan in Rule: fis1033(A,2,B);
Suggestion of plan in Rule: inf1007(A,2,B);
Suggestion of plan in Rule: mat1157(A,2,B).
```

Notice that in this case the mechanism recommends the plan of operation INF1403 because that is the plan with the *lowest* in-char support for the rule that has been performed by a B-case. Finally, the insights and information obtained in the decision mining can also be leveraged for the generation of goal-inference rules.

3.6 Summary of the chapter

In this chapter we discuss the motivation for leveraging automated planning for the Process Mining of unstructured processes. We also relate the Process Discovery task of Process Mining to the definition of a Conceptual Model domain representation, and how it can be extracted from a *log*. We show an example in our educational domain use case. With the discovered Conceptual Model representation at hand, we proceed to perform the discovery of typical patterns via *typical plan mining* (Section 3.3.5). This completes the *process discovery* task, with the capturing of actual executions of the process. Furthermore, we relate the *conformance checking* task to *plot replay* and *model fitness* metrics approaches in the planning paradigm. Finally, we investigate possibilities of *model enhancement* and how they relate to our approach based on a plan-verification and plan-recognition over a conceptual model. The implementations of the algorithms described in this chapter are part of the Library of Typical Plans for Process Mining, discussed in the next chapter.

4 Library of Typical Plans for Process Mining

In this chapter we describe the Library of Typical Plans for Process Mining in which we implement the methods described in the previous chapter[1]. We will discuss the algorithms, referring to parts of the code that implement them when necessary, with examples from simplified domains, custom built to showcase and clarify the workings of the algorithms.

The first section of this chapter describes the plan mining and the composition of the Library Index structure used for the process discovery discussed in Section 3.3.5. The second section describes the algorithms for replay and conformance checking described in Section 3.4. The third section describes experiments implementing the approaches for model enhancement described in Section 3.5.

For the following sections we will use a simplified example domain, with the following configuration:

```
% discipline(Cod, LPrereqs). % Registers a discipline
% Activity clauses
discipline(i1, []).
discipline(j1, []).
discipline(i2, [[i1]]).
discipline(j2, [[j1]]).
discipline(i3, [[i2, j2]]).
discipline(k, []).
discipline(l, []).
discipline(m, [  [i1], [j1] ]).
discipline(n, []).
discipline(z, []).
group(group_nz, [n, z]).
```

The operations are defined as discussed in Section 4.3. The case termination operations are as follows:

```
operation(  grad(Student,Sem), [student/o, semestre/in]).
added(  grad(Student,Sem), grad(Student) ).
deleted( grad(Student,Sem), student(Student)).
precond(grad(Student,Sem), (  app(Student,i1,Si1),  Si1 #=< Sem ,
                              app(Student,j1,Sj1),  Sj1 #=< Sem ,
```

[1] The Prolog code implementing the Library of Typical Plans for Process Mining is available at the author's academic page at PUC-Rio (http://www.inf.puc-rio.br/~vgottin/thesis.html) also freely upon request.

```
                              app(Student,i2,Si2),  Si2 #=< Sem ,
                              app(Student,j2,Sj2),  Sj2 #=< Sem ,
                              app(Student,i3,Si3),  Si3 #=< Sem ,
                              app(Student,k,Sk),     Sk #=< Sem ,
                              app(Student,l,Sl),     Sl #=< Sem ,
                              app(Student,m,Sm),     Sm #=< Sem ,
                              app(Student,
                                group_nz,Sg_nz),  Sg_nz #=< Sem   ) ).

operation( drop(Student, Sem),  [student/o, semestre/in]).
added(drop(Student, Sem), dropped(Student) ).
deleted( grad(Student,Sem), student(Student)).
precond(drop(Student,Sem), (  rep(Student,D,S1),
                              rep(Student,D,S2),     S1 #< S2,
                              rep(Student,D,S3),     S2 #< S3    ) ).
```

A visual representation of the relation of pre-requisites among disciplines and the case termination operations is given in Figure 4:

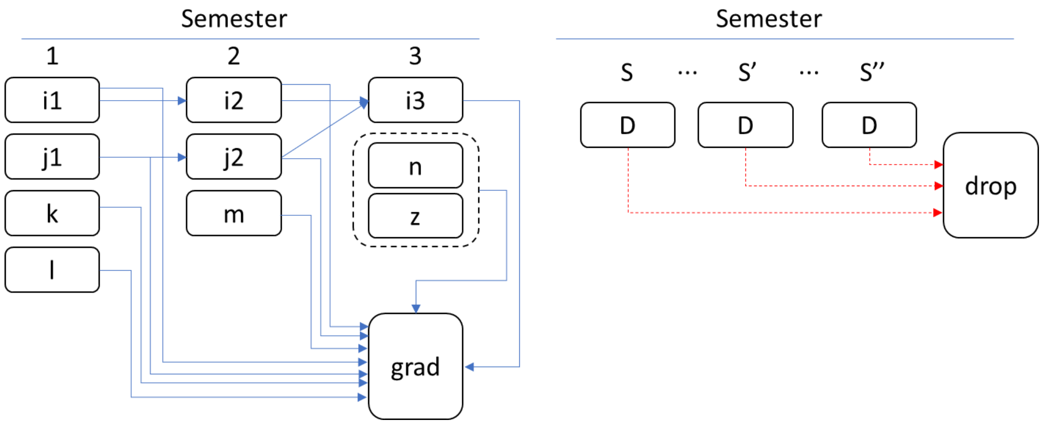


Figure 4 A visual representation of the pre-requisites for disciplines and case termination operations in the short example.

The blue lines represent the success; and the dashed red lines represent discipline failures. In each of the subsections we will consider different sets of plots and goal-inference rules.

4.1 Plan mining algorithms

In this section we discuss the implementation of the plan-mining mechanism via plan-recognition. The methods described here extend the methods in (FURTADO e CIARLINI, 2001), without loss of functionality. We’ll start discussing the most basic mechanisms and proceed to discuss how they are used to compose the Library of Typical Plans for Process Mining. We discuss in Section 4.1.5 the characteristics of the Library of Typical Plans for Process Mining that are not used for the process mining tasks and implement features and capabilities of the methods in (FURTADO e CIARLINI, 2001).

4.1.1 Plan-recognition

The method for plan recognition examines a plot in relation to a goal-inference rule.

Assume a plot $\hat{c}$ of case c, and a goal-inference rule R defined by its agent c, a situation s and a goal g. Our goal is to identify a plan p that is a subsequence of $\hat{c}$ such that the situation s holds before p and the goal g holds after p. This is achived by the algorithm `get_plan`.

In order to define the `get_plan` algorithm we start by defining an auxiliary algorithm `holds_plot` that relates a *plot* and a *plot state* to a *subplot*.

A *plot* is a sequence of ground operations $[\mathrm{s0}, o_1, \dots, o_k]$, $k \geq 1$. A *plot state* is a term or conjunction of terms representing a plot state π, comprised of positive or negative facts. A *subplot* is sequence $[\mathrm{s0}, \dots, o_j]$, $1 \leq j \leq k$ such that π *holds* at the state brought by the execution of the operations in σ.

A few notes on the notation we use in the following discussion: the state π_σ is the state reached by executing a sequence of operations σ. The sequence σ_π is a sequence of operations that reach state π. The sequence $\overleftarrow{\sigma}$ is the reverse sequence of σ.

The algorithm leverages a version of a conventional `holds` meta-predicate - as in (FURTADO e CIARLINI, 2001). A fact f holds after an operation o executed at a state π_σ if either:

(1) o is the pseudo-operation s0 and f belongs to the initial state;

(2) the preconditions of o hold at π_σ and f is a positive effect of o; or

(3) f already held at π_σ and f is not a negative effect of o.

The holds meta-predicate relies on the definitions of the operations (and their `added` and `deleted` clauses). For the examples, consider the following plots:

```
plot(a50, [s0, % Student a50
     i1(a50, 1, 0), j1(a50, 1, 100), k(a50, 1, 100),       % 1st semester
     m(a50, 2, 0),  i1(a50, 2, 100), l(a50, 2, 100)  ]).   % 2nd semester
plot(a51, [s0, % Student a51
     i1(a51, 1, 0),   j1(a51, 1, 100), k(a51, 1, 100),     % 1st semester
     i1(a51, 2, 100), l(a51, 2, 100),  m(a51, 2, 100) ]).  % 2nd semester
plot(a52, [s0, % Student a52
     i1(a52, 1, 0),   j1(a52, 1, 100), k(a52, 1, 100),      % 1st semester
     m(a52, 2, 100),  i1(a52, 2, 100), l(a52, 2, 100)  ]).  % 2nd semester
```

A visual representation of the plots of these students is given in Figure 5:

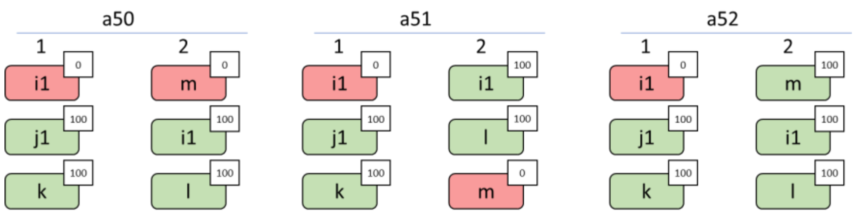


Figure 5 A visual representation of a set of students' plots.

In this representation each box represents a discipline attempt by a student. We highlight failed disciplines in red. The grades obtained by the student in the discipline are shown in square attached to the box. The disciplines are horizontally aligned by semester; and vertically aligned within each semester by the order in which they appear in the event log (and, by consequence, in the trace and in the plot).

As an example, consider the plot given by trace $\hat{c}$ of student a50. For plot state π:

```
student(S), app(S, j1, 1)
```

the `holds_plot` algorithm will determine a subplot:

```
[s0, i1(a50, 1, 0), j1(a50, 1, 100) ] % Subplot of a50
```

Notice that the `holds_plot` algorithm may not find – and therefore, not determine - any suitable subplots. In our Prolog implementation that means the predicate implementing the algorithm simply fails without results, as typical in the logic programming paradigm.

Consider for example the case of student a50 and the plot state π:

```
 app(S,i1,Semester), not rep(S,_Disc,Semester)
```

The algorithm will check all operations in the plot and find that the state does not hold. Upon considering the operation `i1(a50,2,100)` that satisfies the first fact in the plot state, the Semester variable will unify with value 2 – and the second part of the conjunction will fail, since a50 has indeed failed discipline m.

The `holds_plot` algorithm also accounts for the 'clobbered' and 'undone' facts from operations representing events with the same $timestamp$ argument. The operations in the plot are *ordered* with respect to the timestamp, but our implementation accounts for multiple concurrent operations in the same time.

Consider the plot given by the trace of student a51, identical to the one of student a50 but with operations recorded in another order – discipline `m` is now recorded last.

In the straightforward implementation of the algorithm the plot state π holds at the time when operation `i1(a51, 2, 100)` is considered – at that point, the failing operation `m(a51, 2, 100)` has not yet been considered.

Hence, we adapt the method to consider the following.

- We check if the last operation o_j in the candidate subplot $\dot{\sigma}$ has an argument related to a $timestamp$ attribute.
    - If it does not, the subplot $\sigma = \dot{\sigma}$. This guarantees compatibility of the Library of Typical Plans for Process Mining for non-process mining applications.
    - If the last operation o_j in the candidate subplot $\dot{\sigma}$ has a timestamp, we check the remaining operations $[o_{j+1}, \dots o_k]$ in the plot and compose a sequence $[o_{j+1}, \dots o_t]$ such that o_t is the last operation in the remaining operations with that same timestamp.
    - Then, for each term in the conjunction π, we check operations $[o_{j+1}, \dots o_t]$ to see if any of them have positive effects that 'clobber' a negated fact in π or negative effects that 'undo' positive facts in π.
        - If that is the case, the candidate subplot $\dot{\sigma}$ is discarded.

In our example of student a51 above, we obtain `[s0, i1(a51, 1, 0), j1(a51, 1, 100), k(a51, 1, 100), i1(a51, 2, 100)]` as a candidate subplot. The last operation `i1(a51, 2, 100)` is timestamped, so we check the remaining operations `[l(a51, 2, 100), m(a51, 2, 100)]` in the plot. We find that `m(a51, 2, 100)` clobbers the negative fact `app(S,i1,Semester), not rep(S,_Disc,Semester)`, at this point unified as `app(a51,i1,2), not rep(a51,_Disc,2)`. Hence the candidate subplot is discarded.

With the `holds_plot` algorithm we define the `get_plan` algorithm as follows.

Given:

- a plot (from a trace $\hat{c} = [s0, o_1, o_2, \dots, o_k]$ of case c) and a goal inference rule $R = a, s, g$
- we obtain a *subplot* σ_s to the situation s and a *subplot* σ_g to the goal g with the `holds_plot` algorithm.
    - We consider a candidate plan $\dot{p}$ composed by the operations in the subplot to g that are not in the subplot to s. That is, $\dot{p} = [o_{j+1}, \dots, o_l]$ when $\sigma_s = [s0, \dots, o_j]$ and $\sigma_g = [s0, \dots, o_l]$, for $1 \leq j < l \leq k$.
    - The candidate plan is refined to yield a representative plan by a filtering process, as in (FURTADO e CIARLINI, 2001), in which only the operations that are *relevant* to the goal g.

Consider the example of goal-inference rule 5 below, representing the goal of students that failed in a discipline in the first semester of succeeding in that discipline without failing in any others in the second:

```
% Rule 5 – correct early failure
gi_rule( 5, student(Student),
       (   student(Student), rep(Student,Disc1, 1) ),
       (   app(Student, Disc1, 2),
           not rep(Student, _Disc2, 2)
       ) ).
```

For student a50 and a51 we will obtain no plans – notice that the situation in the goal-inference rule is the situation of our previous examples, for which no subplots were found for these students. Consider however student a52, identical to student a50 but more successful, passing in discipline i1 in the second semester.

In this case we obtain the candidate plan `[j1(a52,1,100), k(a52,1,100), m(a52,2,100), i1(a52,2,100)]`.

We refine the candidate plan via an invocation of the auxiliary algorithm `extract_plan`., straightforwardly reimplemented from (FURTADO e CIARLINI, 2001). That algorithm receives as input a sequence of operations σ and a plot state π. In this case, we invoke the algorithm passing the candidate plan $\dot{p}$ and the goal g as input.

The algorithm `extract_plan` operates backwards.

- For each operation o in $\overleftarrow{\sigma}$ we check:
  (1) if one of the positive effects of o is a positive fact in π; or
  (2) if one of the negative effects of o is a negative fact in π.
    - If not, o is discarded from the plan and the remaining operations of $\overleftarrow{\sigma}$ are checked against π.

- Otherwise, o is kept as part of the plan, the preconditions of o are added to π to obtain a new conjunction of facts π', and the remaining operations of $\overleftarrow{\sigma}$ are checked against π'.

Consider again the example of student a52 above. We extract from the candidate plan the operations that are relevant to the goal:

```
app(a52, i1, 2), not rep(a52, _Disc2, 2)                    % The goal
[j1(a52,1,100), k(a52,1,100), m(a52,2,100), i1(a52,2,100)]    % Candidate plan
```

The operations in $\overleftarrow{p}$ are checked. Operation `i1(a52, 2, 100)` contributes to the goal – its effects include `app(a52, i1, 2)`. The preconditions of operation i1 are added to the goal – in this case, there are none, and the process continues by checking operation `m(a52,2,100)` and the remaining operations. The resulting plan is just the operation `i1(a52,2,100)`.

The plan recognition algorithms above are used in the composition of the Library Index, described in Section 4.1.3.

4.1.2 Plan generalization

Besides the plan-recognition algorithms above, we also leverage algorithms for comparing and generalizing plans. As discussed in Section 3.3.5, we resort to two methods of generalization. In this section we describe the algorithms that implement the first one, based on the *most specific generalization* of similar plans. We build upon the methods defined in (FURTADO, 1992; FURTADO e CIARLINI, 2001).

The simplest case is the most specific generalization of two simple plans, given by single operations. Since these are defined in the dynamic schema, we operate only over the operation signature. We rely on the `msg` algorithm defined in (FURTADO, 1992; FURTADO e CIARLINI, 2001), that works as follows.

The algorithm operates over two terms T1 and T2. It yields a generalized term M and builds a list G of generalization substitutions [(E1,E2,V), (E1',E2',V'),…,(E1*,E2*,V*)] where E1, E1',…E* are sub-terms of T1; E2, E2', …, E2* are sub-terms of T2; and V',V'',…,V* are new unique variables. The generalized term will have V replacing both E1 and E2, V' replacing both E1' and E2' in M, and so forth. The algorithm is initiated with an empty list of substitutions M, and the cases are processed are as follows:

- If T1 and T2 are the same variable V, add a term (V,V,V) to the list of substitutions M. The result is V itself.
- If T1 is a variable and T2 is not; or if T2 is a variable and T1 is not, add a term (T1,T2,V) to M. The generalization is a new variable V.
- If T1 and T2 are both not variables and are not of similar types, add a term (T1,T2,V) to M. The result is a new variable Vl. Two terms are similar if they are the same atom or if they are terms with the same functor and same number of arguments.
- If T1 and T2 are both the same atom, no generalization is required, and the result is the atom itself.
- IF T1 and T2 similar types compounded of the form

  T1::=`<functor>(<arg1,arg1’,…,arg1*>`

  and

  T2::= `<functor>(<arg2,arg2’,…,arg2*>`

  With the same functor, we recursively invoke the algorithm for each pair (`arg1, arg2`) and the current list of substitutions M. The result is the generalized term built with `<functor>` and arguments given by the result of the recursive invocation.

In (FURTADO, 1992) the algorithm is described in detail, including an additional step in which the resulting list of substitutions M is checked for unnecessary substituions, replacing new variables introduced by the method by variables already in the original terms when appropriate. We replace that step with one of our own, that deals with non-discriminative arguments, introduced shortly.

The other case is more complex. We deal with the generalization of complex plans, as given by composite operation’s `component` and `dependencies` lists. The first are comprised of tagged operations `<fid>:<op-signature>`, and the latter of lists of `<fid>-<fid>` elements.

The most specific generalization of two complex plans is performed by the algorithm `msg_plan`. The algorithm consider the lists of components C1 and C2 and the lists of dependencies D1 and D2, from complex plan operations P1 and P2, respectively.

The algorithm `msg_plan` will only consider the generalization of two complex plans P1 and P2 that have the same number of components C1 and C2.

It will also consider only complex plans P1 and P2 that have the same number of dependency terms in the standard format described in Section 3.3.5.

The algorithm in (FURTADO e CIARLINI, 2001) also checks for the number of possible substitutions in each list of components – requiring that both complex plans have the same number of possible substitutions. A possible substitution is given by the number of distinct arguments in the components of the plan, counting co-designated variables only once. For example, the number of possible substitions in `f1:g(a, b), f2:h(a, a), f3:i(b, c)]` is three – a, b and c are distinct arguments in the operations. The number of possible substitutions in `[f1:g(A,A),f2:h(A,A),f3:i(A,B)]` is two – only variables A and B are distinct values.

In our implementation we modify this check to only consider *discriminative arguments* in the component operations. Discriminative arguments are those that abide by the variable co-designation and non-co-designation mechanism. Non-discriminative arguments are introduced to represent arguments that we allow to have different values in operations without characterizing different events.

Consider, for example, the case of two complex plan operations P1 and P2 whose components are given by C1 = `[f1:i1(a13,1,73),f2:j1(a13,1,88)]` and C2 = `[f1:i1(a14,1,73),f2:j1(a14,2,88)]`, respectively. The number of possible substitutions in the first one is 4, referring to the distinct values `88`, `73`, `1` and `a13`. The number of possible substitutions in the second one is 5, referring to the distinct values `88`, `73`, `2`, `1` and `a13`. These are very similar plans and yet they would not be considered for the generalization via our first method because student `a14` delayed discipline `a13` by one semester.

Hence, we relax the restrictions over the generalization mechanism and determine that operation arguments that are value entities in the static schema are not *discriminative*.

We define an auxiliary algorithm `discriminative_arguments_plan` that, relates a list of components in standard format C to a list CD of *components in discriminative format* and a list CV of *non-discriminative value components*.

The `discriminative_arguments_plan` algorithm:

- parses C, and
- for each `<fid>-<op-signature>` checks each argument of the operation for the respective entity.
- For each entity that a `value` entity, an element `<value>/<val>` is added to a list `<non-disc-argsvalueslist>`, where `<val>` is the value of the

argument associated to the `<value>` entity. The list CV will contain one element `<fid>-<non-disc-argvalueslist>` for element in C.

- Finally, an element `<fid>:<disc-op-signature>` is added to CD, where `<disc-op-signature>` is the same as `<op-signature>` except only figuring discriminative arguments.

For example, `discriminative_arguments_plan` of our plan P1 will relate C1 = `[f1:i1(a13,1,73),f2:j1(a13,1,88)]` to a list CD1 = `[f1:i1(a13),f2:j1(a13)]` and a list CV1 = `[f1:[semester/1,grade/73], f2:[semester/1,grade/88]]`. The algorithm $discriminative\_arguments\_plan$ for plan P2 will relate C2 = `[f1:i1(a14,1,73),f2:j1(a14,2,88)]` to a list CD2 = `[f1:i1(a14),f2:j1(a14)]` and a list CV2 = `[f1:[semester/1,grade/73], f2:[semester/2,grade/88]]`.

The list CD respective to each plan is then used for the check for the number of possible substitutions, as in the original algorithm. In the example, plan P1 yields list CD1 = `[f1:i1(a13),f2:j1(a13)]` and plan P2 yields CD2=`[f1:i1(a14),f2:j1(a14)]` that have the same number of possible substitutions– one each. Hence, the check succeeds.

This relaxed mechanism allows us to consider the most specific generalization of more diverse, yet still similar under our domain definitions, complex plans. We will leverage this to capture ranges of constraints over the arguments that are non-discriminative later on.

At this point in the `msg_plan` algorithm, we mostly performed preliminary consistency checks. If all checks succeed, we find all possible generalizations of the two plans P1 and P2 via an auxiliary algorithm `msg_matching_plans`. We adapt the original algorithm in (FURTADO e CIARLINI, 2001) to account for the discriminative arguments and to perform constraint boundary expansion.

The algorithm `msg_matching_plans` receives as input the list of dependencies, the list of components in discriminative format and the list of non-discriminative value components of both P1 and P2 – that is, D1, D2, CD1, CD2, CV1 and CV2 as per our definitions above. It relates those to a generalized plan MSG, a list of substitutions SL and a number of constrained-value variables expanded NE. It is processed as follows:

- performs checks of co-designation and non-co-designation of variables, including a retagged of the `<fid>` terms in all component lists;
- expands the bounds of the non-discriminative arguments in CV1 and CV2 via the `boundary_expansion` algorithm described below;
- generalizes the (retagged) lists of *components in discriminative format* via the `msg` algorithm as CD;
- re-applies the constraint expanded varibles to the generalized lists of components.

We start by describing the checks. The algorithm will only generate results for matching plans, meaning that a lifted copy of P1 can be unified under co-designation and non-co-designation constraints with P2. As in (FURTADO e CIARLINI, 2001), "Co-designation (or, respectively, non-co-designation) allows (forbids) the occurrence of the same value (constant or variable) in different parameter positions", and we verify that the order dependencies are the same via a renaming of the tags that makes both sets of dependencies identical.

After these checks, we perform the constraint boundary expansion of the variables in the lists of non-discriminative value components CV1 and CV2. We know, at this stage, that the plans are similar in structure. Furthermore, we have renamed the order-depency tags `<fid>` to match in the lists of components.

Hence, we define an algorithm *expand_bounds* that will process the lists CV1 and CV2 matching the elements `<fid>-<non-disc-argvalueslist>` in each by the same `<fid>`. Each element in `<non-disc-argvalueslist>` is a list of `<value>/<val>`. We ultimately consider for the constraint boundary expansion the matching terms `<val>`, given by a matching `<value>` entity in the matching `<fid>` of each plan. For example, in our current case the lists CV1 = `[f1:[semester/1,grade/73], f2:[semester/1,grade/88]]` and CV2 = `[f1:[semester/1,grade/73], f2:[semester/2,grade/88]]` (whose `<fid>` tags were incidentally already matching before the renaming described above) will be compared element-wise: the first element `[semester/1,grade/73]` is the same in both lists, and will result in no expansions in this example. The second element is `[semester/1,grade/88]` for CV1 and `[semester/2,grade/88]` for CV2 – will focus on it for the examples below.

The constraint boundary expansion is done as follows. Let v1 and v2 be matching `<val>` arguments according to the criteria above (that is, both v1 and v2 are related to the same `<value>` entity in the same `<fid>` component). In our example, we compare the `semester/1` in the second element of CV1 to the `semester/2` in the second element of CV2, hence v1 = 1 and v2 = 2.

We obtain the infimum and supremum values of v1 and v2 as inf1, sup1 and inf2, sup2. The infimum or supremum value of a concrete numerical value is itself. The infimum *inf*(X) (or supremum *sup*(X)) of a variable X is the minimum (maximum) value that can be attributed to it consistently in the current state of the constraints in the domain. For example, if variable X has been previously determined to be constrained between 1 and 10, *inf*(X)=1 and *sup*(X)=10. More complex cases are dealt with by the constraint programming mechanisms, with the atoms `inf` and `sup` standing for the infimum and supremum of the entire domain. For example, determining X>Y and X<10 yields inf(X)=*inf* and sup(X)=10. Determining X>Y, X<10 and Y>5 yields inf(X)=7 and sup(X)=9. These examples and the implementation are all relative to the library of constraint logic programming over finite domains CLP(FD) available for the SWI-Prolog distribution.

We obtain I as the minimum between inf1 and inf2 and Q as the maximum between sup1 and sup2. The minimum between any value and the atom `inf` is the atom `inf`. The maximum between any value and the atom `sup` is the atom `sup`. Then we rely again on the constraint mechanism to define the generalization Q of v1 and v2 with the constraints I =< Q =< M. In our example with v1=1 and v2=2, we have straightforwardly I=1 and M=2. Hence, the generalization of v1 and v2 becomes a variable Q with the constraint annotation (Q in 1 … 2).

The result of the boundary expansion process over the lists of non-discriminative value components CV1 and CV2 in our examples will yield a generalized list CV with constraint-annotated variables. In our example, the resulting list is CV = `[f1:[semester/1, grade/73], f2:[semester/Q, grade/88]], Q in (1..2),` and the number of expanded variables NE = 1.

All the constraint manipulations are handled as *annotated variables* in the CLP(FD) library. We only convert the constraints to an *explicit* term in order to store them reliably as part of the Library Index entries. The algorithm `msg_matching_plans` proceeds to invoke the `msg` algorithm for the lists of components in discriminative format CD1 and CD2, obtaining a generalization CD and the list of substitutions LS.

The algorithm `msg_matching_plans` completes the generation of one alternative generalization for plans P1 and P2 by unifying variables in CD with the variables in CV, effectively applying the expanded constraints over the variables in CD.

Recall that the algorithm `msg_plan` finds all the possible generalizations of matching plans. The resulting *most-specific generalization* is chosen as the one with the minimum number of substitutions.

We'll illustrate the application of the algorithm with some higher-level examples. Assume the following cases, for students a1 and a2:

```
plot(1,  [   s0,
        i1(a1, 1, 100),  j1(a1, 1, 100),  k(a1, 1, 100),  l(a1, 1, 100),
        i2(a1, 2, 100),  j2(a1, 2, 100),  m(a1, 2, 100),
        i3(a1, 3, 100),  n(a1, 3, 100),   grad(a1,3)  ] ).
plot(2,  [   s0,
        i1(a2, 1, 100),   j1(a2, 1, 100),   k(a2, 1, 100),
        l(a2, 2, 100),    i2(a2, 2, 100),  j2(a2, 2, 100),
        m(a2, 3, 100),    i3(a2, 3, 100),
        n(a2, 4, 100),    grad(a2,4)   ] ).
).
```

The plots of these students are illustrated in Figure 6. In this figure, as before, the disciplines in each plot are aligned horizontally according to the semester.

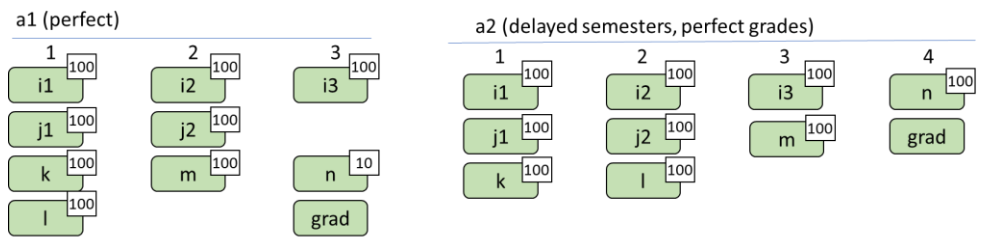


Figure 6 A visual representation of some of the plots of students a1 and a2 in the short example.

Assume that the entire plot of each student has been collected as a complex plan, defining composite operations with the following components and dependencies:

```
% Plan 1: c1(a1)
components : [ f1:i1(a1,1,100), f2:j1(a1,1,100), f3:k(a1,1,100), f4:l(a1,1,100),
f5:i2(a1,2,100), f6:j2(a1,2,100), f7:m(a1,2,100), f8:i3(a1,3,100),
f9:n(a1,3,100), f10:grad(a1,3) ]
```

```
dependencies : [f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-f10,f8-f10,f9-
f10]

% Plan 2: c2(a2)
components : [ f1:i1(a2,1,100), f2:j1(a2,1,100), f3:k(a2,1,100), f4:l(a2,2,100),
f5:i2(a2,2,100), f6:j2(a2,2,100), f7:m(a2,3,100), f8:i3(a2,3,100),
f9:n(a2,4,100), f10:grad(a2,4)]

dependencies : [f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-f10,f8-f10,f9-
f10]
```

In this case, the check for the same number of components (10 in both) and order dependency terms (also 10 in both) succeeds. We obtain the components in discriminative format:

```
% Plan 1: c1(a1)
components in discriminative format: [f1:i1(a1), f2:j1(a1), f3:k(a1), f4:l(a1),
f5:i2(a1), f6:j2(a1), f7:m(a1), f8:i3(a1), f9:n(a1), f10:grad(a1)]

non-discriminative value components: [f1:[semestre/1,nota/100],
f2:[semestre/1,nota/100], f3:[semestre/1,nota/100], f4:[semestre/1,nota/100],
f5:[semestre/2,nota/100], f6:[semestre/2,nota/100], f7:[semestre/2,nota/100],
f8:[semestre/3,nota/100], f9:[semestre/3,nota/100], f10:[semestre/3]]

% Plan 2: c2(a2)
components in discriminative format: [f1:i1(a2), f2:j1(a2), f3:k(a2), f4:l(a2),
f5:i2(a2), f6:j2(a2), f7:m(a2), f8:i3(a2), f9:n(a2), f10:grad(a2)]

non-discriminative value components: [f1:[semestre/1,nota/100],
f2:[semestre/1,nota/100], f3:[semestre/1,nota/100], f4:[semestre/2,nota/100],
f5:[semestre/2,nota/100], f6:[semestre/2,nota/100], f7:[semestre/3,nota/100],
f8:[semestre/3,nota/100], f9:[semestre/4,nota/100], f10:[semestre/4]]
```

The check for the same number of possible substitutions in both lists of components in discriminative format succeeds.

We proceed to find all the matching plan substitutions via `msg_matching_plan`. Since the original dependency lists of both plans are the same, the re-tagging and subsequent unification check succeeds straightforwardly.

We expand obtain the generalization of the non-discriminative value components via `expand_bounds` algorithm, obtaining the generalized list of discriminative arguments CV:

```
[f1:[semestre/1,nota/100], f2:[semestre/1,nota/100], f3:[semestre/1,nota/100],
f4:[semestre/A,nota/100], f5:[semestre/2,nota/100], f6:[semestre/2,nota/100],
f7:[semestre/B,nota/100], f8:[semestre/3,nota/100], f9:[semestre/C,nota/100],
f10:[semestre/D]], % with constraints  (A in 1 .. 2),  (B in 2 .. 3),  (C in 3 .. 4),
(D in 3 .. 4)
```

The invocation of `msg` with the lists of components in discriminative format yields the generalization:

```
[f1:i1(A), f2:j1(A), f3:k(A), f4:l(A), f5:i2(A), f6:j2(A), f7:m(A), f8:i3(A),
f9:n(A), f10:grad(A)]
% With substitutions
[(a1,a2,A)]
```

This is the only generalization found via `msg_matching_plans`. Hence, we select it as the most-specific generalization for this case.

In the next section we will describe how this first method of generalization – via similarity and most-specific generalization – is applied in the composition of the Library Index. The generalization of plans via `msg_plans` will impact, for example, on the *cases* that are considered to follow the generalized plan.

4.1.3 The Library Index

The Library Index is a structure that records the typical plans discovered in the domain, indexing them by the goal-inference rule used for mining. The structure of the Library Index is as follows:

```
<library_index> ::= [<library_entry>] | [<library_entry>, <library_index>]

<library_entry> ::= <rule-id>, plans: [ <rule-planslist ]
<rule_planslist> ::= < plan> | <plan>, <rule-planslist>
```

The definition of the types of `<plan>` is discussed in Section 3.3.5.

Hence, the Library Index is a structure that relates each goal-inference rule (represented by its rule-id) to a list of plans. We use the rule-id as a placeholder for the identification of the rule and report purposes – in the algorithms and mechanisms the rule is actually indexed by a triple (a, s, g) defining the agent, situation and goal of the rule, respectively.

We discussed the overall process of the composition of the library in Section 3.3.5. Here we will discuss the algorithm `build` that implements the general method for building the Library Index.

The `build` algorithm is an entry-point for the main `build_library` algorithm, described below. Before invoking the `build_library`, the `build` algorithm performs a series of checks and initializations, asserting that a domain is correctly specified, and either initializing a new Library Index or managing the state of a current Library Index, if one exists.

The checks include an assertion that a domain has been correctly defined with available cases and at least one goal-inference rule.

The mechanism for dealing with a pre-existing Library Index structure is expanded from (FURTADO e CIARLINI, 2001) to optionally consider only *new* goal-inference rules, ignoring those already in Library Index. If no Library Index exists, all rules are considered new. At the end of this step, the Library Index will contain one element `<rule-id>, plans: [ ]` for each new goal-inference rule available.

For each plot and each goal-inference rule the `build_library` algorithm will, iteratively, by backtracking:

1) Extract a plan with from the plot, relying on the algorithms described in Section 4.1.1;
2) Perform a matching process of the extracted plan and the plans already in the Library Index, which consists of:
   a) Adding the extracted plan to the Library Index if it contains no plans for that entry;
      i) As a simple plan, if the extracted plan is a single operation from the dynamic schema;
      ii) Defining a new composite operation, otherwise;

   or

   b) Generalizing one similar operation in the Library, relying on the algorithms described in Section 4.1.2;

   or

   c) Creating a new generic operation associated to the entry, if it does not contain one; and
   d) Adding the extracted plan to a generic operation complex plan in the entry, with the extracted plan as a specialization;
      i) As a simple plan, if the extracted plan is a single operation from the dynamic schema;
      ii) Defining a new composite operation, otherwise.

In all cases, we update the records of the *cases* for the respective entry when we modify the Library Index. We'll discuss some of the algorithms that manage the state of the Library below.

Step 1 is straightforwardly performed via the `get_plan` algorithm.

Step 2.a implicitly includes a check for existing operations in the index – if there are none, step 2.a is executed. Adding a simple plan – comprising a single operation from the dynamic schema, in step 2.a.i – is straightforward.

For adding complex plans, in step 2.a.ii, an operation signature and frame must be composed. In (FURTADO e CIARLINI, 2001) a process for asking the user for the definition of a complex operation is applied. In our intended process mining applications it is important to make the process as automatic as possible. Hence, we additionally implement an optional mechanism for the automatic generation of complex operation signatures and frames.

The mechanism creates new operation names as required, and determines the arguments in the operation signature, and corresponding operation frame, based on the definitions of the arguments and entity types in the conceptual model. For simplicity, we choose the straightforward approach of only representing the discriminative arguments (see the previous Section) in the complex operation signature. Hence, in our examples, the complex operation signatures are always unary predicates with the first argument corresponding to the `<case-id>`, and the operation frame is, correspondingly, `[student/o]`.

If the Library Index already contains a similar plan, we attempt the first-level generalization via the `msg_plan` algorithm. If a most-specific generalized plan is obtained, we update the corresponding Library Index entry to comprise that generalized plan, updating the *cases* as appropriate.

An additional check is performed – the most-specific generalized plan only substitutes the plan in the Library if it is more general. That is, if the number of variables in discriminative arguments of the new generalized plan is greater – or if a variable expansion has been found.

If the extracted plan is not similar enough to the existing plans to allow for the generalization via `msg_plan` we look for a generic operation in the Library Index entry. If one does not exist, it is created. A similar mechanism for the automatic definition of the operation signature and frame is used as above. Finally, the extracted plan is added to Library Index entry, comprising a specialization of the generic operation in that entry.

We will exemplify the algorithms described above (and auxiliary algorithms) with examples. To illustrate the process, we'll consider the cases of students a1 an a2 from above and the additional following cases:

```
plot(3,   [   s0,
        i1(a3, 1, 70),    j1(a3, 1, 70),   k(a3, 1, 70),  l(a3, 1, 70),
        i2(a3, 2, 70),    j2(a3, 2, 70),   m(a3, 2, 100),
        i3(a3, 3, 70),     n(a3, 3, 70),   grad(a3,3)  ] ).
plot(4,   [   s0,
        i1(a4, 1, 100),  j1(a4, 1, 100),  k(a4, 1, 100),  l(a4, 1, 100),
        i2(a4, 2, 100),  j2(a4, 2, 100),  m(a4, 2, 100),
        i3(a4, 3, 100),   z(a4, 3, 100),  grad(a4,3)  ] ).
plot(5,   [   s0,
        j1(a5, 1, 100),   i1(a5, 1, 100), k(a5, 1, 100), l(a5, 1, 100),
        j2(a5, 2, 100),   i2(a5, 2, 100), m(a5, 2, 100),
        i3(a5, 3, 100),   n(a5, 3, 100),  grad(a5,3) ] ).
plot(6,   [   s0,
        i1(a6, 1, 70),    j1(a6, 1, 70),  k(a6, 1, 70),  l(a6, 1, 70),
        i2(a6, 2, 70),    j2(a6, 2, 70),  m(a6, 2, 100) ] ).
plot(7,   [   s0,
        i1(a7, 1, 70),    j1(a7, 1, 70),  k(a7, 1, 70),
        l(a7, 2, 70),     i2(a7, 2, 70),  j2(a7, 2, 70) ] ).
plot(8,  [   s0,
        i1(a8, 1, 70),   j1(a8, 1, 70),   k(a8, 1, 70),   l(a8, 1, f),
        l(a8, 2, 70),    i2(a8, 2, 70),   j2(a8, 2, 70),  m(a8, 2, 100) ] ).
plot(9,  [   s0,
        i1(a9, 1, 100),  j1(a9, 1, 100),  k(a9, 1, 100),  l(a9, 1, 0),
        l(a9, 2, 100),   i2(a9, 2, 100),  j2(a9, 2, 100), m(a9, 2, f),
        m(a9, 3, 100),   i3(a9, 3, 100),  n(a9, 3, 0),
        n(a9, 4, 100),   grad(a9,4) ] ).
```

A visual representation of the plots of these students is given in Figure 7 and Figure 8. In these figures, as before, the disciplines in each plot are aligned horizontally according to the semester. Within each semester the order of disciplines represents the order of terms in the plot.

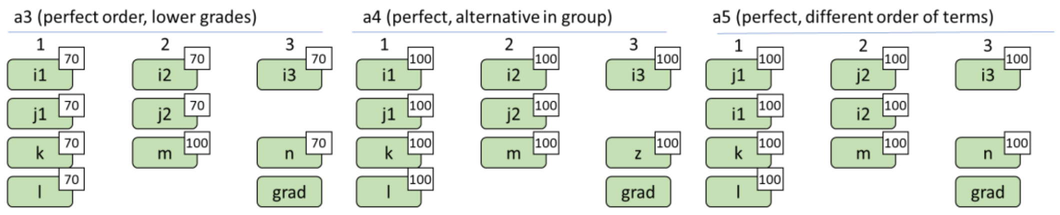

a3 (perfect order, lower grades)
a4 (perfect, alternative in group)
a5 (perfect, different order of terms)
1
2
3
i1
j1
k
l
i2
j2
m
i3
n
z
grad
70
100

Figure 7 A visual representation of some of the students in the short example.

These students, along with a1 and a2, are students who graduated without failing disciplines. They perform different sets of disciplines, in different orders and different grades.

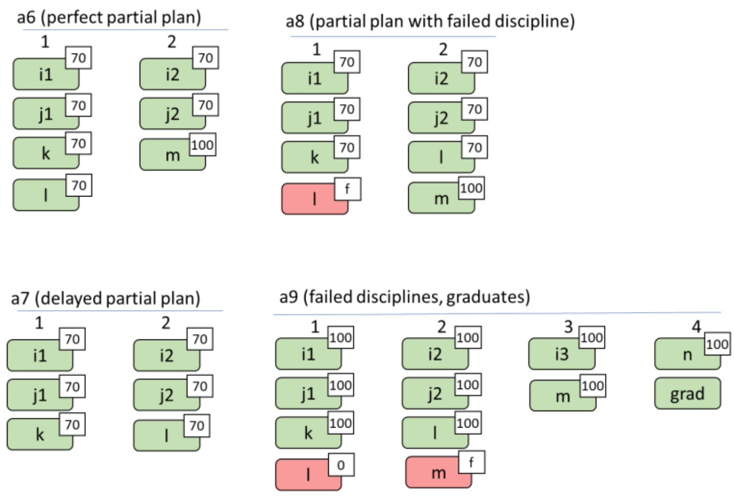

a6 (perfect partial plan)
a8 (partial plan with failed discipline)
a7 (delayed partial plan)
a9 (failed disciplines, graduates)
1
2
3
4
i1
j1
k
l
i2
j2
m
i3
n
grad
70
100
f
0

Figure 8 A visual representation of other students in the short example.

In this set we have partial traces, as well as students with failed disciplines.

The process for building the Library Index recursively selects a plot and a goal-inference rules; finding the plans that satisfy the rule and adding them to the Library Index in the appropriate entry.

We'll consider the example of goal-inference rule 1.

```
gi_rule( 1,
        student(Student),
        (   student(Student),
            not app(Student, _, _), not rep(Student,_, _),
            not grad(Student) ),    not drop(Student) )
        (   not rep(Student,_, _), grad(Student)  )
).
```

This rule represents the goal of new students of graduating without any failed disciplines.

We invoke the `build` algorithm for this set of students and this single goal-inference rule. After the checks and the initialization of the Library Index structure, we have:

```
% Library Index
[ (1,plans:[] ) ] % Goal-inference 1 – no plans
```

And we proceed to the `build_library` algorithm. The algorithm first selects the plot of student `a1`, introduced in Section 4.1.2, and then rule 1 (the only one available).

The plan-recognition via the `get_plan` algorithm will yield the students entire plot as a plan:

```
[i1(a1,1,100), j1(a1,1,100), k(a1,1,100), l(a1,1,100), i2(a1,2,100),
j2(a1,2,100), m(a1,2,100), i3(a1,3,100), n(a1,3,100), grad(a1,3)]
```

The matching process will identify that no plans exist in the Library Index entry. A new composite operation `c1(a1)` is defined and added to the Library Index:

```
[ ( 1, plans:
    [  complex:
       [  composite:c1(a1),
          frame:[student/o],
         components:[f1:i1(a1, 1, 100), f2:j1(a1, 1, 100), f3:k(a1, 1, 100),
f4:l(a1, 1, 100), f5:i2(a1, 2, 100), f6:j2(a1, 2, 100), f7:m(a1, 2, 100),
f8:i3(a1, 3, 100), f9:n(a1, 3, 100), f10:grad(a1, 3)],
         dependencies:[f1-f5, f1-f7, f2-f6, f3-f10, f4-f10, f5-f8, f6-f8, f7-
f10, f8-f10, f9-f10],
         cases:[1]
      ]
    ]
  )
]
```

At this point, a representation of the plans in the entry 1 of the Library Index is given in Figure 9:

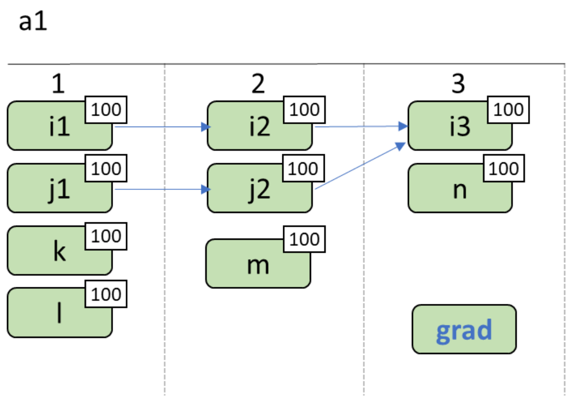


Figure 9 A visual representation of the plan extracted for goal-inference rule 1 and student a1 in the short example.

In the figure above the order dependencies are represented as directed arrows (e.g. the order dependency term `f1-f5` is represented by the arrow pointing from `i1` to `i2`). The order dependencies to the case termination operation are omitted.

Because this is the only plan in the Library Index entry, no second-level generalization is attempted. No other plans are available for the student, and we proceed to check the next plot.

The plot of student a2 is selected. The extracted plan is again the entire plot of the student.

Since a similar plan already exists in the Library, we attempt to generalize both via `msg_plan`. The example of the generalization of the plans for student a1 and a2 is exactly the example used in Section 4.1.2. The generalization succeeds and the composite operation c1 is generalized as follows:

```
    composite:c1(a1),
    frame:[student/o],
    components:[f1:i1(A,1,100),  f2:j1(A,1,100),  f3:k(A,1,100),  f4:l(A,B,100),
f5:i2(A,2,100),  f6:j2(A,2,100),  f7:m(A,C,100),  f8:i3(A,3,100),
f9:n(A,D,100),  f10:grad(A,E) ],
% with constraints clpfd:(B in 1 .. 2), clpfd:(C in 2 .. 3), clpfd:(D in 3 .. 4), clpfd:(E
in 3 .. 4)
    dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-f10,f8-f10,f9-
f10],
    cases:[2,1]) ]
```

Notice that the constraints resulting from the boundary expansion in the generalization are still annotated variables according to the CLP(FD) library implementation. In order to store the new plan, we perform the process of making those constraints explicit, as discussed in Section 4.1.3. The operation c1 in the Library Index entry is substituted by the generalized version above, with explicit constraints.

A visual representation of the plans in the entry 1 of the Library Index is given in Figure 10. The operations with dashed outline are disciplines (or case termination operation `grad`) that can be executed in one of many semesters, delimited by the dashed whisker lines:

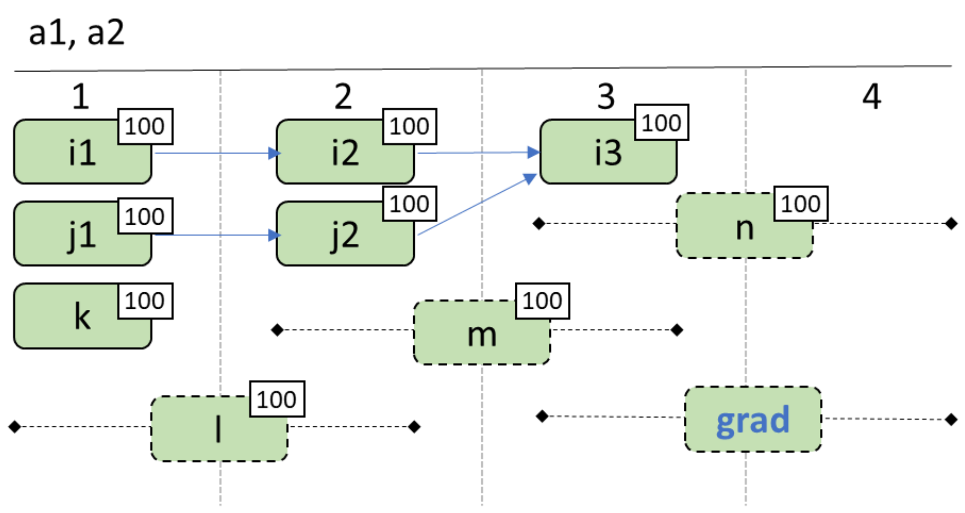


Figure 10 A visual representation of a generalized plan in the short example.

Because there’s still only a single plan in the Library Entry, we still don’t consider the second-level generalization.

The example above illustrates how the Library of Typical Plans for Process Mining identifies that two plans of the same activities with similar ordering relations – but *different timestamps* – are captured as a unique plan for further analysis.

With no more available plans for student a2, we proceed to check the plans for student a3. The plan from student a3 is very similar to that of student a1, except that she obtains lower grades while still being approved in all disciplines. Again, the plan extracted comprises the whole plot of the student.

The first-level generalization via `msg_plan` succeeds yet again, this time expanding the boundaries for the grades obtained by the students that perform the plan. The resulting generalized plan is illustrated in Figure 11. The constraint ranges for the grades for disciplines are given by dashed boxes and minimum-maximum grades:

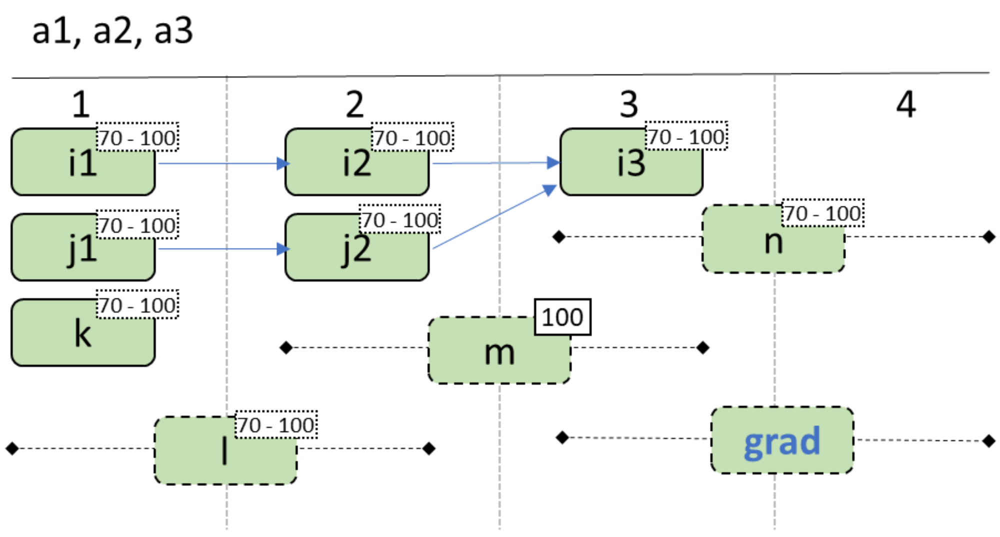


Figure 11 A visual representation of the plan extracted for goal-inference rule 1 and students in the short example.

This example shows that the plan-recognition algorithm is capable of identifying that a ground plan, with concrete values for timestamps, is considered similar to a generalized plan as long as the constraints over the timestamps hold. It also illustrates that the additional non-discriminative arguments are generalizable as well.

Once again, after the consideration of the plans of student a3, there is still only one plan in the Library Index entry and the second-level generalization is not considered. The algorithm proceeds to check the plans for student a4.

Student a4's plot is also very similar to a1's. However, she performs a different discipline to obtain approval in the 'group' of disciplines – a4 performs discipline Z instead on N. Once again, the extracted plot is the student's whole plot.

This time, however, due to the strict requirements for considering plans similar, we find that the plan for student a4 is *not* similar to the generalized c1 in the Library Index. Hence, we compose a new operation c2 and add it to the Library Index:

```
[ ( 1, plans:
        [   complex:(
                composite:c2(a4),
                frame:[student/o],
                components:[f1:i1(a4,1,100),  f2:j1(a4,1,100),  f3:k(a4,1,100),
f4:l(a4,1,100),  f5:i2(a4,2,100),  f6:j2(a4,2,100),  f7:m(a4,2,100),
f8:i3(a4,3,100),  f9:z(a4,3,100),  f10:grad(a4,3) ],
                dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
                cases:[4],
                constraints:[]
            ), % End of complex plan
            complex:(
                composite:c1(A),
                frame:[student/o],
                components:[f1:i1(A,1,B),  f2:j1(A,1,C),  f3:k(A,1,D),
f4:l(A,E,F),  f5:i2(A,2,G),  f6:j2(A,2,H),  f7:m(A,I,100),  f8:i3(A,3,J),
f9:n(A,K,L),  f10:grad(A,M) ],
                dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
                cases:[3,2,1],
                constraints:[ (B in 70 .. 100),  (C in 70 .. 100),  (D in 70 ..
100),  (E in 1 .. 2),  (F in 70 .. 100),  (G in 70 .. 100),  (H in 70 .. 100),
(I in 2 .. 3),  (J in 70 .. 100),  (K in 3 .. 4),  (L in 70 .. 100),  (M in 3 ..
4) ]
            ) % End of complex plan
        ] % End of Plans of rule 1
      )
]
```

A visual representation of the plans contained in the library at this stage is given in Figure 12:

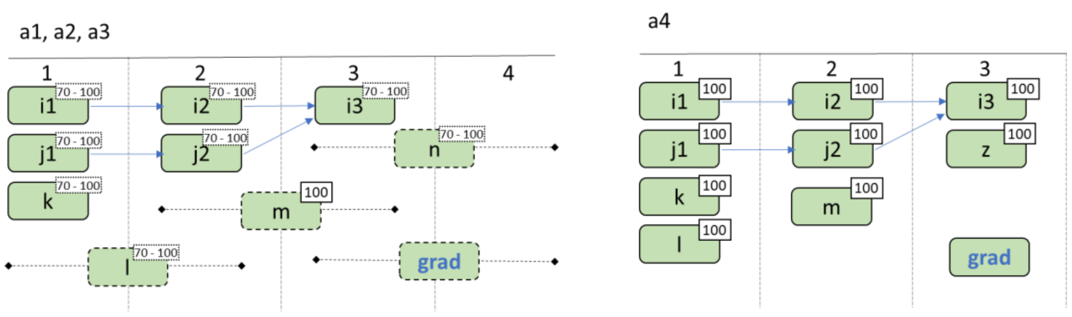


Figure 12 Multiple plans in the Library Index entry in the short example.

Now the Library Index contains more than a single plan for the entry. Hence, the second-level generalization is attempted. There are no generic operations, so a new one is created and added to the index:

```
complex:(generic:g1(A), frame:[student/o], specializations:[c2(A),c1(A)],
cases:[4,3,2,1]
```

The algorithm then checks the plot of student a5 for plans. That plot is identical to a1's, included in the examples to show that the plan-recognition has been adapted do identify that two plots of events with the same timestamps are the same. The examples in 8.1.1 exemplify that in detail. With the addition of the plan extracted for student a5 the final state of the Library is:

```
[ ( 1, plans:
       [ complex:(
             generic:g1(A),
             frame:[student/o],
             specializations:[c2(A),c1(A)], cases:[5,4,3,2,1]
         ), % End of complex plan
         complex:(
             composite:c2(a4),
             frame:[student/o],
             components:[f1:i1(a4,1,100),  f2:j1(a4,1,100),  f3:k(a4,1,100),
f4:l(a4,1,100),  f5:i2(a4,2,100),  f6:j2(a4,2,100),  f7:m(a4,2,100),
f8:i3(a4,3,100),  f9:z(a4,3,100),  f10:grad(a4,3) ],
             dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
             cases:[4],
             constraints:[]
         ), % End of complex plan
         complex:(
             composite:c1(A),
             frame:[student/o],
             components:[f1:i1(A,1,B),  f2:j1(A,1,C),  f3:k(A,1,D),
f4:l(A,E,F),  f5:i2(A,2,G),  f6:j2(A,2,H),  f7:m(A,I,100),  f8:i3(A,3,J),
f9:n(A,K,L),  f10:grad(A,M) ],
             dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
             cases:[5,3,2,1],
             constraints:[ (B in 70 .. 100),  (C in 70 .. 100),  (D in 70 ..
100),  (E in 1 .. 2),  (F in 70 .. 100),  (G in 70 .. 100),  (H in 70 .. 100),
(I in 2 .. 3),  (J in 70 .. 100),  (K in 3 .. 4),  (L in 70 .. 100),  (M in 3 ..
4) ]
         ) % End of complex plan
       ] % End of Plans of rule 1
   )
]
```

Notice that the case 5 (the id in the `initial_state` of student a5) is added to cases of both the composite plan `c1` and the generic plan `g1`.

The state of the library above is the final Library Index obtained by this set of students and the goal-inference rule 1 – students a5, a7, a8 and a9 don't reach the goal. Students a6, a7 and a8 don't graduate, and student a9 fails a discipline. Hence, no plans are extracted for those cases.

4.1.4 Management of the Library Index

The Library Index is a data structure to hold all the plans found for the goal-inference rules considered. The structure of Library has been presented above, and the structure of the plans in the Library Index entries in Section 3.3.5. At all times during the mining process the Library Index structure is stored as an asserted fact `library_index(<library_index>)` in the Prolog database.

Notice that in the asserted representation of the Library – the one we've used to showcase the plans and the Library, except when stated otherwise – the constraints over the variables in the plans are *explicitly stored in a list*. The constraints are said to be in *explicit format* in the asserted `library_index` but, in all the plan-mining algorithms, they are in the *implicit format*, as annotated variables in the CLP(FD) library format. We define an algorithm `library_explicit_implicit` that relates a `<library-index>` structure in *explicit format* (with constraints in explicit format) to a library in *implicit format* (a similar structure in which the plans do not contain `constraint` terms, but the equivalent restrictions are applied over the variables in the CLP(FD) constraint store. The algorithm can be used to obtain a library in *explicit format* from one in *implicit format* and vice-versa.

Thus, the Library is typically not directly accessed via the `library_index` predicate. Rather, we define an algorithm `read_library` that (1) obtains the current Library Index and (2) translates the list to the implicit format, via $librar\_explicit\_implicit$, yielding that as a result.

Because the Library Index is stored as an asserted fact, the methods for updating the plans or including a new plan) in an entry must retract the current fact and assert the modified version. Because we must store the constraints of the new plans in explicit format, we define auxiliary algorithms that also rely on `library_explicit_implicit` for those manipulations as well.

Finally, the format of the Library Index in the Library of Typical Plans for Process Mining is different from the one used in (FURTADO e CIARLINI, 2001). Because of our secondary goal of not discarding any features of the original Library of Typical Plans, we also provide methods to translate a Library Index to (and from) that format. When translating from the Library Index to that format, the information of the cases is discarded – but the constraints are asserted in the constraint store. When translating from that format to the Library Index, no cases will be considered (the `cases` list in all plans will be variable), but any constraints over the variables in the constraint store will be recognized and carried over.

4.1.5 Highlighted additional features

The implementation of the algorithms related to Plan mining in the Library of Typical Plans for Process Mining comprise several novel features with respect to the original BLIB (FURTADO e CIARLINI, 2001). In this section we highlight a few of those main aspects that differentiate the Library of Typical Plans for Process Mining and the BLIB and also discuss the ‘retro compatibility’ of the implementation in other aspects.

The most important novel feature of the Library of Typical Plans for Process Mining relates to the new Library Index structure. The novel format holds explicit constraints over values in the arguments of the operations – according to the definitions of the value entities in the static schema. The structure also holds the *cases* that relate to each plan and, therefore, to each rule.

The mechanisms for generalization are changed to collect the constraints over these values expanding the boundaries of the values already observed in the plan. The generalization algorithms also deal with non-discriminative arguments, which are important for the consideration of whether a generalization took place when a new plot is observed. As previously stated, this relaxed mechanism allows us to consider the most specific generalization of more diverse, yet still similar under our domain definitions, complex plans.

The mechanism for accounting for constraints also comprise the plan-recognition, although in limited scenarios, and the detection of ‘clobbered’ and ‘undone’ facts by leveraging the timestamp argument.

The algorithm for building a (or updating a previously generated) Library Index from plots is also novel. It captures the cases of the respective plans and updates them accordingly in the structure.

Our approach for Process Mining balances the *simplicity* of the model and the other quality criteria. Hence, in the examples and in the general approach described in this thesis we don’t always exploit the full power of the representation allowed by the conceptual model – which is, in fact, according to the Entity-Relationship model.

As a consequence, there are several features of the Library of Typical Plans for Process Mining that stem from the original BLIB (FURTADO e CIARLINI, 2001) and that are not used for the Process Mining examples. In this section we discuss some of this features and possibilities of representation in the adopted conceptual model approach.

The Library of Typical Plans for Process Mining can be used for the same purposes as the original BLIB. In the implementation, the aspects of the original formulation were considered and we strove to maintain all functionalities. With the mechanism for translating a Library Index to (and from) the format used in BLIB we developed a testing mechanism to ensures that all examples concerning richer features generate equivalent results when executed in the Library of Typical Plans for Process Mining.

As discussed in Section 3.3.1 the conceptual model accounts for richer definitions in the static schema. We considered the use of an entity attribute in the examples of model enhancement in Section 3.5 by adding an argument to the `entity` clause. This is a change in notation, in respect to the original formulation in (FURTADO e CIARLINI, 2001), in which attributes are defined by separate `attribute` clauses and the only additional argument for the `entity` clauses define a privileged identifying attribute. The static schema may also additionally comprise multiple entity classes organized in a is-a hierarchy, in which attributes are inherited by sub-classes. In any case, the Library of Typical Plans for Process Mining accounts for (properly defined) entity attributes and multiple entity classes.

The static schema may also define relationships between entity classes. Facts regarding relationships among entity instances can be used in the definition of the situation or goal of goal-inference rules, same as facts regarding entity attributes. This is also demonstrated by the correct replication of the results of the examples in (FURTADO e CIARLINI, 2001), which include relationship clauses and facts.

In the dynamic schema we defined that operations created from domain activities have only positive effects. This is a strong restriction on the representative power of the conceptual mode. However, by relying on the meta-predicate *holds* both the plan-recognition and plan-verification algorithms are general and capable of dealing with operations with negative effects.

Another aspect that is accounted for in the Library of Typical Plans for Process Mining that is not required for the process mining approach is the capacity of dealing with plots involving multiple agents. In the Process Mining fashion, we expect to be able to identify the case uniquely for each event in the log. Thus, it is trivial to separate the traces, and therefore the plots, of each case. This relates especially to the `extract_plan` algorithm – for the process mining approach it wouldn't need to check that the operation refers to the relevant agent. We keep that check for compatibility with the BLIB.

Finally, in (FURTADO e CIARLINI, 2001) three kinds of operations are defined for re-structuring the index after a batch of plots are considered. These `rebuild` algorithms recognizes in the index – and update it accordingly – multi-level generalizations; generic operations as components of composite operations; and multi-level compositions. Again, we strove the maintain the retro-compatibility of the Library of Typical Plans for Process Mining and adapted those methods to deal with the Library Index. Hence in several cases, our resulting Library Index can be subjected to these re-structuring processes and updated as expected. The re-structuring of the Library Index was not the focus of our analyses, however, and we expect that the investigation of Library Index `rebuild` algorithms will comprise future work, especially interesting for novel approaches of model enhancement.

4.2 Conformance Checking algorithms

Recall the discussion in Section 3.4. We define a plan-verification method for the conformance checking via replay of plots. The algorithm for plan-replay is described in the following section. The algorithm for support-counting and other metrics of interest are discussed in Section 4.2.2.

4.2.1 Replay

The plan-verification `forward_check` algorithm is as follows. It relates a plot $\hat{c} = [s0, o_i, \ldots, o_k]$ to a list F of failed operations (and their precondition sets) and a list A of failed operations *allowed* in the plot. The algorithm will produce all combinations of A and F from the plot by backtracking.

The `forward_check` algorithm starts by splitting the trace into two lists: one representing the sequence of successfully executed previous operations $\varepsilon = [s0]$, initially comprising only the initial state operation s0; and a sequence $\sigma = [o_i, \dots, o_k]$ comprising the remaining operations in the plot. Then, it invokes the `forward_check_acc` algorithm that deals with the recursive cases.

In the following, let $\overline{\sigma^{i,j}}$, for $1 \leq i, j \leq k$ be the subsequence of σ starting in operation o_i and ending in operation o_j. In this definition, k is the number of operations in the original plot and o_k is the last operation. For ease of representation, let $\overline{\sigma^{i}}$ be the subplot of σ starting in operation o_i. and ending in operation o_k.

The `forward_check_acc` will consider each operation o_i in σ, roughly as follows:

1. If σ is the empty sequence, the algorithm terminates with an empty list A and an empty list F.
2. Otherwise, we check whether o_i has preconditions.
    a. If it has no preconditions or if at least one alternative conjunction of preconditions is satisfied by the successfully executed previous operations, o_i is conformant in the domain – we recursively invoke the algorithm with the remaining plot, and considering o_i a successfully executed operation. The algorithm yields the results of that recursive invocation.
    b. Otherwise, we consider the operation *discrepant*. The algorithm yields the results of two recursive invocations, by backtracking:
        i. One in which we do not consider o_i as a successfully executed operation and proceeding checking the remainder of the plot – thus ignoring the effects of that operation;
        ii. Another in which we *do* consider o_i to be a successfully executed operation, while also adding it to a list of 'allowed' operations.

A more detailed description of the algorithm, matching the predicates that implement it in our Prolog implementation, is given in the following.

In step 1, if σ is the empty sequence, the algorithm terminates with an empty list A and an empty list F.

Otherwise, in step 2, we check whether o_i has preconditions. This check comprises accounting for sets of alternative preconditions as defined by the `precond` clauses of that operation defined in the dynamic schema. Recall the definition of Π as the set of alternative preconditions of o_i, as discussed in Section 3.3.2.2.1.

In step 2.a, the operation o_i has no preconditions, $\Pi = \{\emptyset\}$, or at least one alternative conjunction of preconditions $\pi \in \Pi$ is satisfied in the state π_ε (the state resulting from the subplot ε). These are the cases in which we consider the operation *conformant* in the domain. We determine this in a similar manner as in the plan mining methods, relying on the meta-predicate `holds`. In this case, we obtain a new list of previously executed operations ε' by adding o_i to ε. We then recursively invoke the `forward_check_acc` algorithm with ε', the list of remaining operations in the plot $\overline{\sigma^{i+1}}$. The algorithm yields the resulting lists A and F from the recursive invocation.

In step 2.b none of the alternative preconditions $\pi \in \Pi$ of o_i hold in π_ε, and we consider the operation *discrepant*.

In step 2.b.i. we obtain a list F1 of failed operations by the recursive invocation of `forward_check_acc` for sequence $\overline{\sigma^{i+1}}$ with the same ε. The algorithm yields a list comprising o_i and the elements in F1 as failed operations, and the list A obtained from the recursive invocation.

In the alternative recursive invocation of the algorithm reached via backtracking in step 2.b.ii we 'allow' the discrepant operation o_i. In this invocation we consider the sequence $\overline{\sigma^{i+1}}$, and a list of executed operations ε' obtained by adding o_i to ε, and a list A' obtained by adding o_i to A. Also, in this case the algorithm yields the resulting list F as the list of failed operations.

Consider the following cases for an example:

```
% Ok - partial trace. M requires either I1 or J1.
plot(95,
    [   s0,
        i1(a95, 1, 100), %<--- this satisfies
        m(a95, 2, 100)  %<-- this
    ]
).

% Drops for no reason - it drops because of a known reason
% If drop is forgiven, nothing else.
initial_database(96, student(a96)).
plot(96,
    [   s0,
        i1(a96, 1, 100),
        j1(a96, 1, 100),
        k(a96, 1, 100),
        l(a96, 1, 100),
        i2(a96, 2, 100),
        drop(a96,3) %<--- dropped without satisfying preconditions
    ]
).

% Similar to a96
% Drops without satisfying full condition -
% only part of the preconditions of the drop are satisfied.
initial_database(96-2, student(a96-2)).
plot(96-2,
    [   s0,
        i1(a96-2, 1, 0),
        i1(a96-2, 2, 0),
        i1(a96-2, 3, 100),
        drop(a96-2,3)
    ]
).

% Student that *does* conform - it drops because of a known reason!
initial_database(97, student(a97)).
plot(97,
    [   s0,
        i1(a97, 1, 0),
        i1(a97, 2, 0),
        i1(a97, 3, 0),
        drop(a97,3) %<-- satisfied!
    ]
).

% This student does not perform I1.
% That means I2 happens even though it shouldn’t. Same for I3.
% M does not appear in the failed list because J1 suffices.
initial_database(98, student(a98)).
plot(98,
    [   s0,
        %i1(a98, 1, 100), <-- Does not perform I1
        j1(a98, 1, 100),
        k(a98, 1, 100),
        l(a98, 1, 100),
        i2(a98, 2, 100), %<-- shouldn’t – no I1
        j2(a98, 2, 100),
        m(a98, 2, 100), %<--- *can* perform this because of j1
        i3(a98, 3, 100), %<-- shouldn’t - I2 required for I3.
        n(a98, 3, 100),
        grad(a98,3)
    ]
).

% Does not conform - graduates without most disciplines.
% Should not be able to perform J2 - it requires I1
initial_database(99, student(a99)).
plot(99,
    [   s0,
        i1(a99, 1, 100),
        j2(a99, 2, 100),
        grad(a99,2)
    ]
).

% Nonconforms - With I1 out of order, I2, and  I3 and grad shouldn’t happen
% Notice M does not fail - j1 happens before
initial_database(100, student(a100)).
plot(100,
    [   s0,
        j1(a100, 1, 100),
        k(a100, 1, 100),
        l(a100, 1, 100),
        i2(a100, 2, 100), %<--- Should fail because i1 is not there
        j2(a100, 2, 100),
        m(a100, 2, 100), % <-- Should succeed out of j1
        i1(a100, 3, 100), %<--- Out of order
        i3(a100, 3, 100), %<--- Fails because i2 fails
        n(a100, 3, 100),
        grad(a100,3)
    ]
).
```

In the examples we will ignore the collection of the preconditions of the discrepant operations for ease of representation.

We'll first consider the plot of student a95. The algorithm `forward_check` splits the plot in the two lists and invokes algorithm `forward_check_acc` with $\varepsilon = [s0]$ and $\sigma =$ [i1(a95, 1, 100), m(a95, 2, 100)].

In (1)`forward_check_acc`[2], with $\varepsilon = [s0]$ and $\sigma =$ [`i1(a95, 1, 100), m(a95, 2, 100)`]: we consider the first operation in $o_1 =$ `i1(a95, 1, 100)`. It has no preconditions, so we consider it *conformant*. We obtain ε'= [`s0, i1(a95, 1, 100)`] and $\overline{\sigma^2}$ = [`m(a95, 2, 100)`], and invoke the algorithm recursively.

In (2) `forward_check_acc`, with $\varepsilon =$ [`s0, i1(a95, 1, 100)`] and $\sigma =$ [`m(a95, 2, 100)`] we consider operation $o_2 =$ `m(a95, 2, 100)`. This operation has two alternative preconditions: $\Pi =$ [[`app(a95,i1,A),A<2`],[`app(a95,j1,B),B<2`]]. We find that the conjunction $\pi =$ (`app(a95,i1,A),A<2`) holds in the plot ε. Hence, the operation is also conformant. The algorithm is recursively invoked again

In (3) `forward_check_acc`, $\sigma =$ [ ] and the algorithm yields an empty list A and an empty list F; Thus (2), (1), and the initial invocation of `forward_check` also yield A = [ ] and F = [ ]. There are no more solutions, and the algorithm ends. This result means that the plot is entirely conformant.

The results for this call are, therefore:

```
Case 95
% The allowed operations
[]
%The failed operations
[]
```

[2] We precede the name of the algorithm with (n) for the n-depth recursive invocation.

We'll now consider the case of student a96. The algorithm `forward_check` splits the plot in the two lists and invokes algorithm (1)`forward_check_acc` with $\varepsilon = [s0]$ and $\sigma =$ [`i1(a96,1,100), j1(a96,1,100), k(a96,1,100), l(a96,1,100), i2(a96,2,100), drop(a96,3)`]. The first few recursive invocations are all straightforward – disciplines I1, J1, K and L don't have pre-requisites.

We skip ahead to (5)`forward_check_acc` with $\varepsilon =$ [s0, `i1(a96,1,100), j1(a96,1,100), k(a96,1,100), l(a96,1,100)`] and $\sigma =$ [`i2(a96,2,100), drop(a96,3)`]. The operation $o_5 =$ `i2(a96,2,100)` has a single alternative precondition $\Pi =$ [[(`app(a96,i1,A),A<2`)]]. The single precondition conjunction $\pi =$ (`app(a96,i1,A),A<2`) is satisfied by ε.

In (6)`forward_check_acc` we have $\varepsilon =$ [s0, `i1(a96,1,100), j1(a96,1,100), k(a96,1,100), l(a96,1,100), i2(a96,2,100)`] and $\sigma =$ [`drop(a96,3)`]. We check operation o_6= [`drop(a96,3)`] and find that is has a precondition set $\Pi =$ [[(`rep(a96,A,B), rep(a96,A,C), B #< C, rep(a96,A,D), C #< D`)]]. This condition cannot be met: the operation is *discrepant*. Hence, this scenario spawns two continuations.

In the first one, representing the continuing execution of the plot with a failed operation, we recursively invoke (7-1)`forward_check_acc` with $\varepsilon =$ [s0, `i1(a96,1,100), j1(a96,1,100), k(a96,1,100), l(a96,1,100), i2(a96,2,100)`] and $\sigma =$[ ]. Notice that the failed operation in (6) is not part of ε.

In (7-1) we obtain F1 = [ ] and A = [ ]. Then (6) yields the same A, and F = [`drop(a96,3)`]. These results are also yielded by the previous recursive calls (5), (4), (3), (2), and (1), up to the original `forward_check`, configuring one final result of the algorithm.

In (6) we make another alternative recursive call. In this one, we *allow* for the failed operation, as if its preconditions had been satisfied.

We invoke (7-2)`forward_check_acc` with $\varepsilon =$ [s0, `i1(a96,1,100), j1(a96,1,100), k(a96,1,100), l(a96,1,100), i2(a96,2,100), drop(a96,3)`] and $\sigma =$[ ]. The call yields A = [ ] and F = [ ]. Then, (6) yields the same F and A=[`drop(a96,3)`]. These results are also yielded by the previous recursive calls (5), (4), (3), (2), and

(1), up to the original `forward_check`, configuring another final result of the algorithm.

The results for this case are:

```
Case 96
% The allowed operations – from call (1)(2)(3)(4)(5)(6)(7-1)
[]
%The failed operations – from call (1)(2)(3)(4)(5)(6)(7-1)
[drop(a96,3)]

% The allowed operations – from call (1)(2)(3)(4)(5)(6)(7-2)
[drop(a96,3)]]
%The failed operations – from call (1)(2)(3)(4)(5)(6)(7-2)
[]
```

We’ll consider, in a higher level, the results for the other students. In the results below we adopt a more concise representation, where each line is an alternative result of `forward_check`. As before, the preconditions that could not be met are suppressed.

```
Case 96-2
% Forgiven > Failed Operations
[]  >  [drop(a962,3)]  % Same as 96 – preconditions of drop partially satisfied.
[drop(a962,3)]  >  []

Case 97
[]  >  []  %Preconditions of drop satisfied – conformant.

Case 98                                             % i1 causes i2 to fail, which
[]  >  [i2(a98,2,100),i3(a98,3,100),grad(a98,3)] % causes i3 to fail.  Notice m
                                                  % does not fail, since j1 succeeds.
[grad(a98,3)]  >  [i2(a98,2,100),i3(a98,3,100)]    % Allowing grad changes nothing.
[i3(a98,3,100)]  >  [i2(a98,2,100),grad(a98,3)]    % Same.
[i3(a98,3,100),grad(a98,3)]  >  [i2(a98,2,100)]
[i2(a98,2,100)]  >  [grad(a98,3)]    % Allowing i2: i3 succeeds
[i2(a98,2,100),grad(a98,3)]  >  []

Case 99
[]  >  [j2(a99,2,100),grad(a99,2)]
[grad(a99,2)]  >  [j2(a99,2,100)]
[j2(a99,2,100)]  >  [grad(a99,2)]
[j2(a99,2,100), grad(a99,2)]  >  []

Case 100
                                                       %i1 out of order, cause i2 to
[]  >  [i2(a100,2,100),i3(a100,3,100),grad(a100,3)] % fail, etc. Since i2 will not
       % be in the ‘correctly executed’ list, i3 fails even though at this time i1 –
       % which caused i2 to fail – has already been executed successfully!
[grad(a100,3)]  >  [i2(a100,2,100),i3(a100,3,100)]
[i3(a100,3,100)]  >  [i2(a100,2,100),grad(a100,3)]
[i3(a100,3,100),grad(a100,3)]  >  [i2(a100,2,100)]
```

```
[i2(a100,2,100)]  >  [] % Allowing i2 to happen before i1 will solve all problems.
```

4.2.2 Support, confidence and interest metrics

In this section we discuss the algorithms that implement the computations of the support and other interest metrics, discussed in Section 3.4.2. We define an algorithm `library_report_support` that computes and reports the support (and other metrics) of the goal-inference rules and plans in the Library Index. The algorithm can optionally compute and report the metrics of a subset of the goal-inference rules.

The algorithm relies on an auxiliary algorithm `rule_report_support` with one argument – a goal-inference rule $r = (a, s, g)$. In practice, we use the goal-inference rule-id as stand-in identifier. The computation is as follows.

We obtain the number of cases in the domain $|\mathcal{C}|$. This is computed as the sum of the number of available plots. Notice that this will consider cases for which no plans have been mined. For example, consider the example of Section 4.1.3. After the `build_library` process we have, for the goal-inference rule 1, plans for students a1, a2, a3, a4 and a5. The domain includes students a6, a7 a8 and a9. The total number of cases $|\mathcal{C}| = 9$.

We obtain the cases RC that follow the rule and a list LOC relating each plan in the rule to a list of cases. This is done via an auxiliary algorithm `rule_cases`.

In the `rule_cases` algorithm we query the library (via `read_library`) for all plans of rule with the id r. We obtain the cases – from the `cases` term – of each plan operation and compose a list LOC of elements `(<op-signature>, <cases>)`. The list LOC contains the operation signatures of all plan operations and the respective list of cases.

For example, for the resulting Library Index entry of goal-inference rule 1 we have above we have LOC= `[(g1(A),[5,4,3,2,1]),(c2(a4),[4]),(c1(A),[5,3,2,1])]`. We also obtain a list RC of cases that is the union of all the `<cases>` in LOC. In our example, RC=`[5,4,3,2,1]`.

After invoking `rule_cases`, we have in `rule_report_support` the number of items in RC as k_r, that is, the number of cases that perform a plan in rule r. The computation of the support($s \Rightarrow g$) = $\frac{k_r}{|\mathcal{C}|}$ is straightforward from there.

In order to compute the confidence($s \Rightarrow g$) of the rule we must find the number of cases that reach the situation s, but not necessarily the goal g, or rule r.

We obtain a set C(s) of cases that reach the situation s through an auxiliary algorithm `cases_reach` that considers each of the cases in the domain and leverages the `holds_plot` algorithm described in Section 4.1.1. We don't need to consider the cases in RC – we already know that they are members of C(s) by construction.

We compute the support(s) as $\frac{\mathrm{C}(s)}{|C|}$, and the confidence($s \Rightarrow g$) of the rule as support($s \Rightarrow g$)/support(s).

For the computation of the lift($s \Rightarrow g$) of the goal-inference rule, for example, we must compute $\frac{\mathrm{support}(s \Rightarrow g)}{(\mathrm{support}(s)\mathrm{support}(g))}$. We have the support($s \Rightarrow g$) and the support($s$) at hand. We can compute the support(g) similarly to the support(s), relying also on the `cases_reach` auxiliary algorithm to obtain C(g).

At this stage, we may also compute the other metrics of interest. We have experimented, for example, with metrics such as the leverage and conviction of the rules, but have not focused on their analysis and further investigation is required to determine whether they are appropriate. Regardless, we posit that the algorithm described above will support the computation of all metrics that depend on these kinds of support values.

Recall the list LOC of operations and their respective cases. For element in LOC we compute the support and confidence of the respective plan (see Section 3.4.2.2). We apply similar computations - except instead of considering the set RC to obtain k_r, we use the set of cases of the Library Index entry operation. For the computation of the confidence of the plan we reuse the support(s), already computed to obtain the confidence of the goal-inference rule.

4.3 Enhancement algorithms

In this section we discuss the implementation of algorithms that are used in the Model Enhancement task, discussed in Section 3.5. In Section 4.3.1 we discuss algorithms for the generalization of goal-inference rules. In Section 4.3.2 we discuss the formulation and algorithms related to decision mining.

4.3.1 Generation of goal-inference rules

In this Section we will discuss algorithms for the (semi-)automatic generation of goal-inference rules as described in Section 3.5.5.

4.3.1.1 Interesting cases

We discussed how we could rely on the plots of interesting students to determine new rules. For example, by leveraging the plot state brought by the student's plot, we could use that state the situation for a new rule.

Upon identifying a case c of interest, we can compose a goal-inference rule a^c, s^c, g where s^c is the situation of case c and goal g is determined depending on the interest of the analysis, typically comprising the effects of a case termination operation. This will generate a rule that will capture the plans that other cases, sharing a characteristic and in the same situation as c, performed to reach the goal. Notice that if we have case attributes (e.g. in the decision mining examples of Section 3.5.6) we can determine that the agent a^c shares characteristics with case c.

Recall that in our notation state π_σ is the state reached by executing a sequence of operations σ. We define an algorithm `plot_state` that yields a plot state π_σ given the plot from a student's trace σ. It is straightforwardly implemented relying on the `holds` meta-predicate.

We will provide examples of this in combination with the methods for the relaxation of rules in the next sections.

4.3.1.2 Lifting variables

We also discussed the *relaxation* of rules, so that they may apply to more cases, via the lifting of the arguments of one or more terms with grounded elements in the situation or goal.

The situation or goal is typically given by π as a conjunction of success or failure clauses. Let $t_i, t_j \in \{s_1, s_2, \ldots, s_n\}$ be both success or failure clauses part of a plot state π of the format defined in Section 4.2:

```
<op-success> ::= <success-functor>(<op-case>,<activity-attr>,<op-id-arglist>).
<op-failure> ::= <failure-functor>(<op-case>,<activity-attr>,<op-id-arglist>).
```

When lifting the variables in π we allow for co-designation $v = v_1 = v_2$ of the lifted variables v_1 in t_1 and v_2 in t_2 if and only if they share the same position as argument in the success or failure clause's arguments. That condition guarantees that the same variable v will be used to refer either to the same *event-identifying attribute* in the `<op-id-arglist>` of both clauses. The same reasoning as above applies for terms $t_i, t_j \in \{g_1, g_2, \dots, g_m\}$ in the goal conjunction g.

We explore a first algorithm for the lifting of plot states `lift_state`. In this configuration, the algorithm allows for the combination of all possible liftings, co-designated or not. The algorithm relates a plot state π to a plot state π' in which one or many variables have been lifted.

The algorithm considers each term t_i in π, lifting the appropriate elements in the `<op-id-arglist>`:

1. In one execution path, it lifts the elements to a predefined variable for that argument, so that the it will be co-designated with other liftings.
2. In another execution path, reached by backtracking, it lifts the elements to a free variable.

In our example domain only the *semester* attribute is part of the `<op-id-arglist>`, so we will only lift that argument.

For example, consider the goal-inference rule 5 in our short example domain:

```
gi_rule( 5, student(Student),
        (   student(Student), rep(Student,Disc1, 1) ),
        (   app(Student, Disc1, 2),
            not rep(Student, _Disc2, 2)           ) ).
```

The results of the `lift_state` algorithm, obtained by backtracking, for the goal g in that rule are:

```
% The resulting lifted state             , the substitutions
(app(A,Disc1,B), not rep(A,Disc2,B)) ,  [(2,B)];       % Both co-designated
(app(C,Disc1,B), not rep(C,Disc2,_)) ,  [(2,B)];       % First co-designated
(app(E,Disc1,_), not rep(E,Disc2,B)) ,  [(2,B)];       % Second co-designated
(app(G,Disc11,_),not rep(G,Disc2,_)) ,  [].            % None co-designated
```

This version of the algorithm generates all the combinations of co-designation and free variables. It does not distinguish the cases that are equivalent – in the example above, co-designating the semester *only* the first or *only* the second term is effectively the same thing. Optimizations could be implemented to avoid generating these results.

However, we opt to instead define a restricted version of the algorithm because – even with a single argument *semester* under consideration - the number of possible combinations of lifted and free variables grows exponentially with the number of terms in π.

For example, consider the case of student a6 in our example of Section 4.1.3. Let σ be the sequence of operation that comprise the student`s plot. We obtain the plot state π_σ resulting from the execution of his plot:

```
student(a6), app(a6, i1, 1), app(a6, j1, 1), app(a6, k, 1), app(a6, l, 1),
app(a6, i2, 2), app(a6, j2, 2), app(a6, m, 2)
```

For this plot state, the algorithm `lift_state` yields:

```
[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,B), app(A,i2,C),
 app(A,j2,C), app(A,m,C)],             [(1,B),(2,C)];     % all co-designated

[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,B), app(A,i2,C),
 app(A,j2,C), app(A,m,_)],             [(1,B),(2,C)]      % m free

[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,B), app(A,i2,C),
 app(A,j2,_), app(A,m,C)],             [(1,B),(2,C)]      % j2 free

( ... ) % 4 results suppressed

[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,B), app(A,i2,_),                     % disciplines in 1st
 app(A,j2,_), app(A,m,_)],             [(1,B)]            % semester co-designated

[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,_), app(A,i2,C),
 app(A,j2,C), app(A,m,C)],             [(1,B),(2,C)]      % l free

[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,_), app(A,i2,C),
 app(A,j2,C), app(A,m,_)],            [(1,B),(2,C)]       % l and m free

( ... ) % 117 results suppressed

[student(A), app(A,i1,_), app(A,j1,_),                   % all free
 app(A,k,_), app(A,l,_), app(A,i2,_),
 app(A,j2,_), app(A,m,_)], []
```

Even lifting only one argument – our example only contains the *timestamp* as an event identifying attribute – we obtain 128 possible combinations of lifting.

This comprises too many for the generation of goal-inference rules, as we discussed in Section 3.5.5. It is likely that domain-dependent definitions should restrict the number of possible combinations of liftings.

For the short example, we determine the `lift_state_restricted` algorithm. It operates similarly to the lift_state algorithm but restricts the lifting to *all* arguments co-designated by position and value, or all free variables. Hence, for the same example, `lift_state_restricted` yields:

```
[student(A), app(A,i1,B), app(A,j1,B),
 app(A,k,B), app(A,l,B), app(A,i2,C),
 app(A,j2,C), app(A,m,C)],             [(1,B),(2,C)];     % all co-designated

[student(A), app(A,i1,_), app(A,j1,_),                  % all free
 app(A,k,_), app(A,l,_), app(A,i2,_),
 app(A,j2,_), app(A,m,_)], []
```

We provide an example that combines the first approach – of using the plot state resulting from a student's plot – and the lifting of variables in a single experiment, in our short domain. Consider the execution of the example in Section 4.1.3. We will simulate a domain-dependent analysis by identifying "noncomplying" students of rule 1. We do so through an auxiliary algorithm `noncomplying_cases` that leverages a (simplified) version of `rule_cases` (as defined in Section 4.2.2) to obtain the plots of all students that *don't* follow a plan in a rule. Let NC be the list of 'noncomplying' cases of a rule. In this example, NC = [a6,a7, a8, a9].

For each case in NC we obtain the plot state π_σ, where σ is the student's plot, and iteratively generate new goal-inference rules $r = a, s, g$ where the situation s is obtained via the lifting algorithm described above, and the goal g is the goal from goal-inference rule 1. In our example, we generate the following rules:

```
% From student a6's plot; free variables in lifting
gi_rule(2 - 1,
        student(A),
       ( app(A,m,_),  app(A,j2,_), app(A,i2,_), app(A,l,_),
         app(A,k,_),  app(A,j1,_), app(A,i1,_), student(A) ),
        ( not rep(A,_,_), grad(A) ) ).

% From student a6's plot, co-designated variables in lifting
gi_rule(2 - 2,
        student(A),
        ( app(A,m,B), app(A,j2,B), app(A,i2,B), app(A,l,C),
         app(A,k,C), app(A,j1,C), app(A,i1,C), student(A) ),
       ( not rep(A,_,_), grad(A) ) ).

% From student a7's plot, free variables in lifting
gi_rule(2 - 3,
        student(A),
        ( app(A,j2,_), app(A,i2,_), app(A,l,_), app(A,k,_),
          app(A,j1,_), app(A,i1,_), student(A) ),
       ( not rep(A,_,_), grad(A) ) ).

% From student a7's plot, co-designated variables in lifting
gi_rule(2 - 4,
        student(A),
        ( app(A,j2,B), app(A,i2,B), app(A,l,B), app(A,k,C),
          app(A,j1,C), app(A,i1,C), student(A) ),
        ( not rep(A,_,_), grad(A) ) ).

% From student a8's plot, free variables in lifting
gi_rule(2 - 5,
        student(A),
        ( app(A,m,_), app(A,j2,_), app(A,i2,_), app(A,l,_),
          rep(A,l,_), app(A,k,_), app(A,j1,_), app(A,i1,_), student(A) ),
        ( not rep(A,_,_), grad(A) ) ).

% From student a8's plot, co-designated variables in lifting
gi_rule(2 - 6,
        student(A),
        ( app(A,m,B), app(A,j2,B), app(A,i2,B), app(A,l,B),
          rep(A,l,C), app(A,k,C), app(A,j1,C), app(A,i1,C), student(A) ),
        ( not rep(A,_,_), grad(A) ) ).
% From student a9's plot, free variables in lifting
gi_rule(2 - 7,
         student(A),
        ( grad(A), app(A,group_nz,_), app(A,n,_), rep(A,group_nz,_), rep(A,n,_),
          app(A,i3,_), app(A,m,_), rep(A,m,_), app(A,j2,_), app(A,i2,_),
          app(A,l,_), rep(A,l,_), app(A,k,_), app(A,j1,_), app(A,i1,_) ),
         ( not rep(A,_,_) grad(A) ) ).
% From student a9's plot, co-designated variables in lifting
gi_rule(2 - 8,
        student(A),
        ( grad(A), app(A,group_nz,B), app(A,n,B), rep(A,group_nz,C), rep(A,n,C),
app(A,i3,C), app(A,m,C), rep(A,m,D), app(A,j2,D), app(A,i2,D), app(A,l,D),
rep(A,l,E), app(A,k,E), app(A,j1,E), app(A,i1,E) ),
        ( not rep(A,_,_), grad(A)) ).
```

The plan-mining process then generates the following Library Index structure:

```
[ ( 1, plans:
       [ complex:(
             generic:g1(A),
             frame:[student/o],
             specializations:[c2(A),c1(A)], cases:[5,4,3,2,1]
         ), % End of complex plan
         complex:(
             composite:c2(a4),
             frame:[student/o],
             components:[f1:i1(a4,1,100),  f2:j1(a4,1,100),  f3:k(a4,1,100),
f4:l(a4,1,100),  f5:i2(a4,2,100),  f6:j2(a4,2,100),  f7:m(a4,2,100),
f8:i3(a4,3,100),  f9:z(a4,3,100),  f10:grad(a4,3) ],
             dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
             cases:[4],
             constraints:[]
         ), % End of complex plan
         complex:(
             composite:c1(A),
             frame:[student/o],
             components:[f1:i1(A,1,B),  f2:j1(A,1,C),  f3:k(A,1,D),
f4:l(A,E,F),  f5:i2(A,2,G),  f6:j2(A,2,H),  f7:m(A,I,100),  f8:i3(A,3,J),
f9:n(A,K,L),  f10:grad(A,M) ],
             dependencies:[f1-f5,f1-f7,f2-f6,f3-f10,f4-f10,f5-f8,f6-f8,f7-
f10,f8-f10,f9-f10],
             cases:[5,3,2,1],
             constraints:[ (B in 70 .. 100),  (C in 70 .. 100),  (D in 70 ..
100),  (E in 1 .. 2),  (F in 70 .. 100),  (G in 70 .. 100),  (H in 70 .. 100),
(I in 2 .. 3),  (J in 70 .. 100),  (K in 3 .. 4),  (L in 70 .. 100),  (M in 3 ..
4) ]
         ) % End of complex plan
       ] % End of Plans of rule 1
   ), % End of rule 1
  ( 2 - 1, plans:
       [  complex:(
             generic:g2(O),
             frame:[student/o],
             specializations:[c4(O),c3(O)],
             cases:[5,4,3,2,1]
         ), % End of complex plan
         complex:(
             composite:c4(a4),
             frame:[student/o],
             components:[f1:i3(a4,3,100),f2:z(a4,3,100),f3:grad(a4,3)],
             dependencies:[f1-f3,f2-f3],
             cases:[4],
             constraints:[]
         ), % End of complex plan
         complex:(
             composite:c3(P),
             frame:[student/o],
             components:[ f1:i3(P,3,Q),f2:n(P,R,S),f3:grad(P,T)],
             dependencies:[f1-f3,f2-f3],
             cases:[5,3,2,1],
             constraints:[(Q in 70 .. 100), (R in 3 .. 4), (S in 70 .. 100),
  (T in 3 .. 4)
         ) % End of complex plan
       ] % End of Plans of rule 2-1
  ), % End of rule 2-1
( 2 - 2, plans:
       [  complex:(
             generic:g3(U),
             frame:[student/o],
             specializations:[c6(U),c5(U)],
             cases:[5,4,3,1]
         ), % End of complex plan
         complex:(
             composite:c6(a4),
             frame:[student/o],
             components:[f1:i3(a4,3,100),f2:z(a4,3,100),f3:grad(a4,3)],
             dependencies:[f1-f3,f2-f3],
             cases:[4],
             constraints:[]
         ), % End of complex plan
         complex:(
             composite:c5(V),
             frame:[student/o],
             components:[ f1:i3(V,3,W),f2:n(V,3,X),f3:grad(V,3)],
             dependencies:[f1-f3,f2-f3],
             cases:[5,3,1],
             constraints:[(W in 70..100),(X in 70..100)]
         ) % End of complex plan
       ] % End of Plans of rule 2-2
   ), % End of rule 2-2
( 2 - 3, plans:
       [  complex:(
             generic:g4(Y),
             frame:[student/o],
             specializations:[c8(Y),c7(Y)],
             cases:[5,4,3,1]
         ), % End of complex plan
         complex:(
             composite:c8(a4),
             frame:[student/o],
             components:[ f1:m(a4,2,100), f2:i3(a4,3,100), f3:z(a4,3,100),
f4:grad(a4,3)],
             dependencies:[f1-f4,f2-f4,f3-f4],
             cases:[4],
             constraints:[]
         ), % End of complex plan
         complex:(
             composite:c7(_),
             frame:[student/o],
             components:[ f1:m(Z,A1,100), f2:i3(Z,3,B1), f3:n(Z,C1,D1),
f4:grad(Z,E1) ],
             dependencies:[f1-f4,f2-f4,f3-f4],
             cases:[5,3,2,1],
             constraints:[ (A1 in 2 .. 3), (B1 in 70 .. 100), (C1 in 3 .. 4),
(D1 in 70 .. 100), (E1 in 3 .. 4) ]
         ) % End of complex plan
       ] % End of Plans of rule 2-3
   ), % End of rule 2-3
( 2 - 4, plans:
       [  complex:(
              composite:c9(a2),
              frame:[student/o],
              components:[f1:m(a2,3,100), f2:i3(a2,3,100), f3:n(a2,4,100),
f4:grad(a2,4)],
             dependencies:[f1-f4,f2-f4,f3-f4],
             cases:[2],
             constraints:[]
         ) % End of complex plan
       ] % End of Plans of rule 2-3
  ), % End of rule 2-3
  (2 - 5, plans : []),
  (2 - 6, plans : []),
  (2 - 7, plans : []),
  (2 - 8, plans : [])
]
```

In the results above, we can see that the Library Index entry for the goal-inference rule 1 is unchanged. The only cases that perform plans of the new rules are again a1, a2, a3, a4 and a5 – that's intuitively correct, since we did not change the goal of graduating without failed disciplines. A visual representation and discussion of the plans obtained for the new rules is given in the following Figure 13:

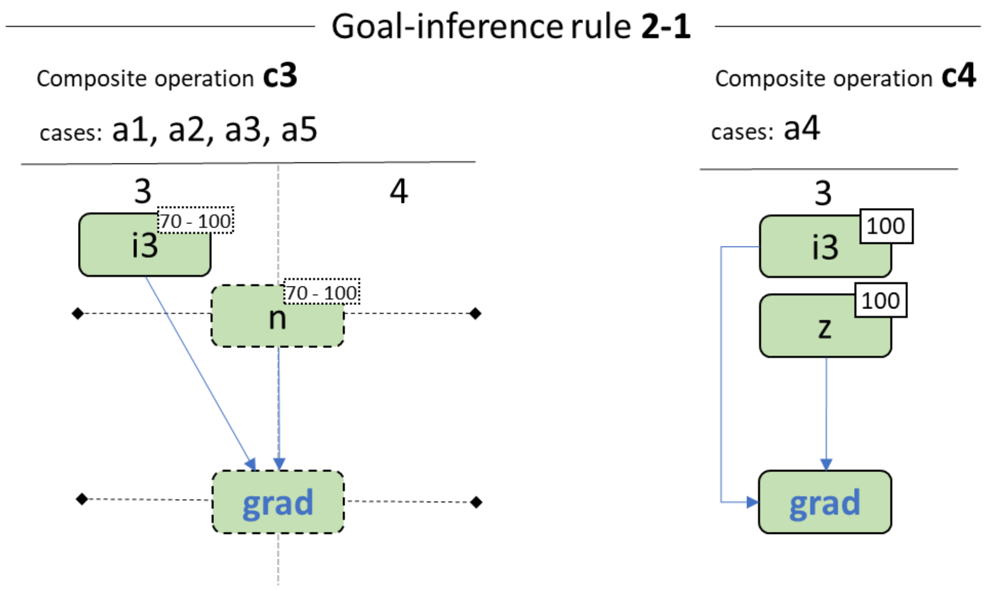


Figure 13 A visual representation of the plans obtained for goal-inference rule 2-1 in the short example.

The goal-inference rule 2-1 was created from the situation of student a6 (recall Figure 8), by lifting the variables indicating the semesters freely. Hence, find plans by all the students a1, a2, a3, a4 and a5 that graduated without failed disciplines. The interest metrics for this rule and the plans are:

```
Support of Rule 2-1:
    support:    0.5556
    confidence: 0.6250
    lift:       1.1250
    leverage:   0.0617
    conviction: 1.1852
        >g2(A):
                support:    0.5556    confidence: 0.6250
        >c4(a4):
                support:    0.1111    confidence: 0.1250
        >c3(A):
                support:    0.4444    confidence: 0.5000
```

The support, as expected, indicates that a little above half the students in the domain perform plans for this rule. The confidence of the rule is rather low – besides students a1, a2, a3, a4 and a5, the situation of the rule is also reached by student a6 (obviously) as well as students a8 and a9 – both eventually approve in the disciplines that a6 was approved in, and the situation does not forbid them from having failed disciplines.

We compare those results to those obtained with rule 2-2. A visual representation is given in Figure 14.

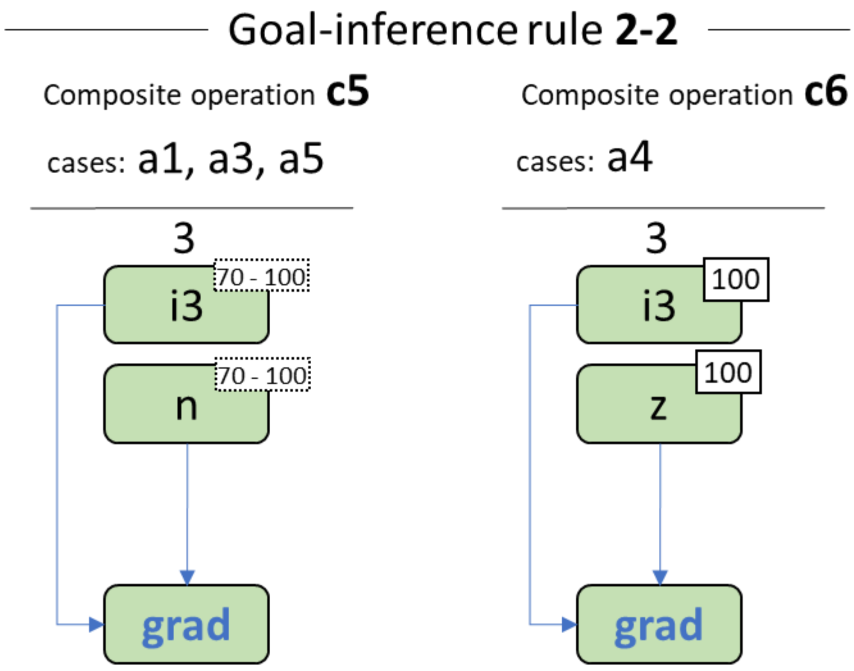


Figure 14 A visual representation of the plans obtained for goal-inference rule 2-2 in the short example.

This rule was also created from the situation of student a6,but the variables introduced in the arguments referring to semesters were co-designated. This means that only plans that performed the sets of disciplines in similar semesters reach the plot state determined by the situation. Most notably, we can see that student a2 *does not* perform the rule. That is intuitively explained – student a6 has not delayed any disciplines, and student a2 has delayed a few.

The interest metrics for the example are:

```
Support of Rule 2-2:
    support:    0.4444
    confidence: 0.8000
    lift:       1.4400
    leverage:   0.1358
    conviction: 2.2222
        >g3(A):
                support:    0.4444    confidence: 0.8000
        >c6(a4):
                support:    0.1111    confidence: 0.2000
        >c5(A):
                support:    0.3333    confidence: 0.6000
```

The support is lower than that of rule 2-1 (with one less student). However, the confidence of the rule is much higher – this means that the rule has captured the plans of all the students that reached student a6's situation, except a6 herself.

The goal-inference rule 2-3 was created from the lifted situation of student a7. A visual representation of the plans is given in Figure 15.

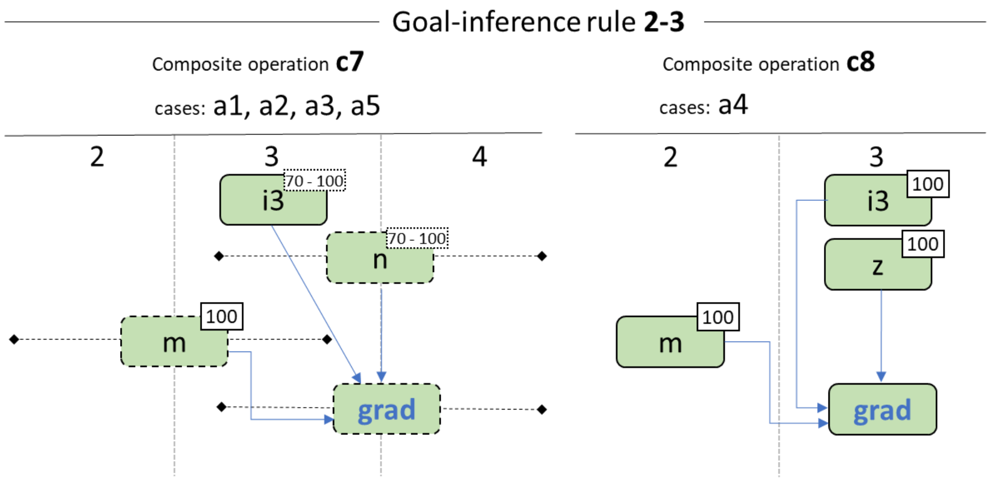


Figure 15 A visual representation of the plans obtained for goal-inference rule 2-3 in the short example.

In comparison with student a6, student a7 has performed one less discipline – discipline `m`. Hence, the plans incorporate that operation. Since for rule 2-3 we have lifted the variables representing the semesters freely, we find that the plans encompass the execution of discipline m in the second semester. This is a consequence of the way the goal was built, but makes little sense in the context of the educational domain – student a7 has (most likely) already finished the second semester, and the plans that would require him to perform additional disciplines in that semester are (contextually) unfeasible. Regardless, the plan denoted by operation c7 can still be performed from semesters 3 and onwards, but this case is an example of how the process generating goal-inference rules might require additional domain restrictions.

We proceed to the example of goal-inference rule 2-4. A visual representation of the resulting plan is given in Figure 16:

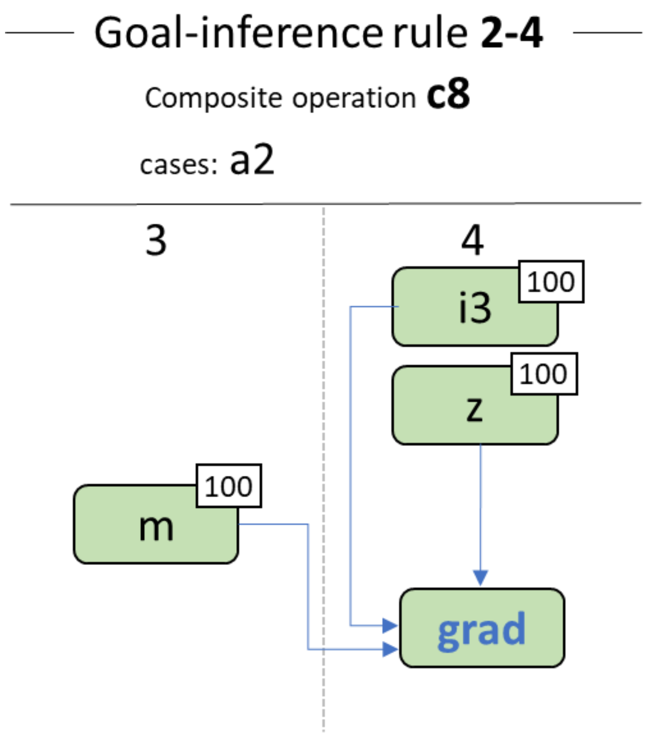


Figure 16 A visual representation of the plans obtained for goal-inference rule 2-4 in the short example.

This goal-inference rule was created by the lifting of the variables in the situation reached by student a7 with co-designated variables for the semesters. Because the student delayed operation `l` from the recommended first semester to the second, students a1, a3 and a5 *do not* reach that plot state. They have all been approved in discipline `l` in an earlier semester than disciplines `i2` and `j2`. The interest metrics for this rule and plan are:

```
Support of Rule 2-4:
    support:    0.1111
    confidence: 0.2500
    lift:       0.4500
    leverage:   -0.1358
    conviction: 0.5926
        >c9(a2):
                support:    0.1111    confidence: 0.250
```

This example illustrates how lifting a rule with co-designated variables might still generate a rule whose plans are infrequent or not very representative. This is reflected by the scores in the interest metrics.

Rules 2-5 and 2-6, generated from the situation reached by student a8's plot, result in *no plans*. This is intuitively easy to see – the situation will include the requirement that a student *failed* in discipline `1`, but the goal requires that the student not fail in any disciplines. The same is true for rules 2-7 and 2-8, generated with the situation reached by student a9's plot, with the additional observation that that situation *includes* that the student is already graduated. Even if it didn't include failed disciplines, the plan-process would find only empty plans for these rules and discard them preemptively. Both of these examples motivate the removal of terms from the clauses with predicated that allow a domain specialist user to guide the removal process. We will discuss algorithms for such in the next section.

4.3.1.3 Removing clauses

In Section 3.5.5 we also discussed how goal-inference rules can be created by the relaxation of requirements via the removal of terms from the situation or goal. Formally, given a rule $r = a, s, g$ we define a new conjunction $s' \subset \{s_1, s_2, \dots, s_n\}$ and compose a new rule $g' = a, s', g$. We can also define a new conjunction $g' \subset \{g_1, g_2, \dots, g_m\}$ and compose a new rule $g' = a, s, g'$, instead. Finally, we can combine both s' and g' in a rule $r' = a, s', g'$. Notice, however, that there will be $(2^n - 1) \times (2^m - 1)$ total combinations of s' and g'. For any reasonably large values of n and m that is impractical.

For this kind of goal-inference rule generation, we define algorithm `relax_conjunction`. It relates a conjunction of terms π, possibly representing a plot state, to a conjunction π' in which one term of π has been removed. The algorithm is very straightforward. When invoked in sequence, it is capable of generation all conjunctions formed by subsets of terms of the original conjunction. However, similarly to the lifting of variables described in Section 4.3.1.2, we also have the problem that there will be too many combinations of removal of terms, and consequently, too many conjunctions π' with which to compose new rules.

To that end, we define auxiliary algorithm `relax_conjunction_restricted`. Besides the conjunction of terms π this algorithm receives as input two lists: one list TK of term functors that dictates terms to *keep*, and one list TR of functors that dictates terms to *remove* from the conjunction. The removal of functors takes precedence. We start by removing from π all terms whose functor matches one of the functors in TR. Then, we remove one element from π whose functor does not match any of the elements in TK. In both cases we consider that the term might be negated in the conjunction, and deal with it accordingly.

Below we provide some examples. Let π be the conjunction (`student(A), not app(A,D1,1), not rep(A,D1,2), not grad(A)`). Invoking the algorithm `relax_conjunction_restricted` for π, TK = [`grad`] and TR=[ ] yields, by backtracking:

```
(not app(A,D1,1), not rep(A,D1,2), not grad(A));
(student(A), not rep(A,D1,2), not grad(A));
(student(A), not app(A,D1,1), not grad(A)).
```

Invoking the algorithm `relax_conjunction_restricted` for π, TK = [`grad, student`] and TR=[`rep`] yields only:

```
(student(A), not grad(A)).
```

The pruning performed by the restrictions seems small at first, but the iterative invocation of the algorithm will severely diminish the exponentially increasing number of possible sub-conjunctions of a conjunction. Still, the numbers resulting from naïve combinations may be too high and could rely on domain knowledge for additional filtering. We provide a discussion on our conclusion in this topic in Chapter 9.

We provide an example in our short example domain. Consider the resulting Index Library from the examples in Section 4.3.1.2. We discussed how rule 2-6 generated no plans due to the fact that student a8 (from whose plot we composed the situation) had failed in a discipline. We'll use that goal-inference rule as an example of the mechanism for relaxing the restrictions of a rule via removal of terms.

We invoke the `relax_conjunction_restricted` for π equal to the situation in 2-6 with TR=[`grad,rep`]. Removing `grad` allows us to capture plans of students that haven't graduated yet. Removing `rep` will allow us to capture plans of students that may have failed on one or more disciplines. We also invoke the algorithm for π equal to the goal in 2-6. In that case, we choose TK=[`grad`]. We use the results, obtained iteratively, to compose new rules. With the restrictions in place, we generate seven alternative rules. We highlight the first one of these rules:

```
% From rule 2-6, with removed terms from situation and goal.
gi_rule(3 - 1,
       student(A),
       ( app(A,j2,B), app(A,i2,B), app(A,l,B), app(A,k,C),
         app(A,j1,C), app(A,i1,C), student(A) ),
      grad(A)  ).
```

The results obtained with this rule are:

```
% A snippet of the resulting Library Index
% (...)
  ( 3 - 1, plans:
       [  complex:(
              composite:c10(F1),
              frame:[student/o],
              components:[ f1:m(F1,3,100), f2:i3(F1,3,100), f3:n(F1,4,100),
f4:grad(F1,4) ],
              dependencies:[f1-f4,f2-f4,f3-f4],
              cases:[9,2],
              constraints:[]
         )
       ]
  ),
%(...)
```

The plan is illustrated in Figure 17.

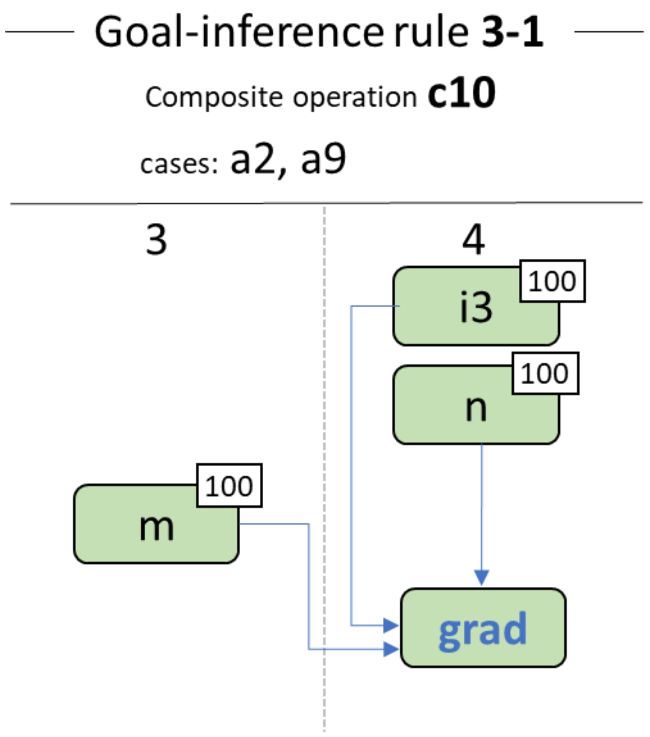


Figure 17 A visual representation of the plans obtained for goal-inference rule 3-1 in the short example.

This example shows that with the lifted restrictions we are able to capture the behavior of student a9 in the plan, whereas before he was ignored due to the failed disciplines in his plot.

Notice that since we used the rule 2-6, which had been previously lifted from a situation obtained from a plot, this example configures an example of all the three methods we proposed in tandem.

4.3.2 Metrics for decision mining

In this section we provide the formulation for the discussion in Section 3.5.6.

Let a set of attributes $\Psi \subset A$ configure the set of case attributes of interest for the analysis. We consider $\Psi = \{a_1, \ldots\}$ to be non-empty for the decision mining task. With $\Psi = \{a_1, \ldots, a_n\}$ where all attributes a_i, $1 \leq i \leq n,$ are categorical, the number of characteristic classes will be equal to the number of unique combinations of values for those attributes. For attributes with numerical, continuous, or too many categorical values it might be necessary to cluster them (in representative bins or classes), and use the bin (or class) instead.

Let $\mathbb{W} = \{W_1, W_2, \ldots, W_k\}$ be the set of characteristic classes for the analysis. Each characteristic class W_j, $1 \leq j \leq k$ determines a unique value for each attribute in Ψ. That is, W_j defines values $w_{j,1}, \ldots, w_{j,n}$ for attributes $a_1, \ldots, a_n$, respectively.

We call W-cases the set of all cases of characteristic class W. That is, $c \in \mathcal{C}$ is a W_j-case iff $\#_{a_i}(c) = w_{j,i}$ for all $1 \leq j \leq k$ and $1 \leq i \leq n$. We further define k_W to be the total number of W-cases. For each $W \in \mathbb{W}$ let $k_{W,r}$ be the number of W-cases that perform a plan of rule r. In the example of rule 9, from previously, we have:

```
Characteristic of Cases in Rule 9:
  Characteristic enroll_a: 11 cases
  Characteristic enroll_b: 10 cases
  Characteristic enroll_c: 1 cases
  Characteristic enroll_d: 1 cases
```

That is, $k_{\mathrm{A},9} = 11$, $k_{\mathrm{B},9} = 10$, $k_{\mathrm{C},9} = 1$ and $k_{\mathrm{D},9} = 1$.

We define the *in-rule support* of W as defined as $\frac{k_{W,r}}{k_r}$, where k_r is the number of cases that perform a plan mined from rule r (see Section 4.2.1).

We define the *in-characteristic support* (or *in-char* for short) as $\frac{k_{W,r}}{k_W}$.

The *in-plan support* is defined as $\frac{k_{W,p}}{k_p}$, where $k_{W,p}$ is the number of cases in k_W that follow the plan p of rule r, and k_p is the number of cases that perform p.

The *in-char* support of the plan is similarly computed. It is defined as $\frac{k_{W,p}}{k_W}$, where $k_{W,p}$ is as above and k_W is again the number of cases in characteristic class W.

Finally, the Jaccard index (or Jaccard similarity coefficient) is defined in our notation as $\frac{k_{W,r}}{k_W+k_r-k_{W,r}}$. The formulation follows from the definition of the Jaccard index as the ratio between the intersection and the union of two sets. We consider the total cases with characteristic W that perform a plan of rule r akin to the *intersection* of cases that perform a plan in the rule, and the cases of that characteristic class.

4.4 Summary of the chapter

In this chapter we define the algorithms that embody the methods described in Chapter 3, and presented the Library of Typical Plans for Process Mining that implements the algorithms. We describe the main aspects of the library, the data structures and the relation to prior work in Section 4.1, along with the algorithms for plan mining. In Section 4.2 we define the algorithms for the conformance checking task of process mining – both the replay and computation of interest metrics. Finally, in Section 4.3 we define the algorithms for the enhancement task. All are given with examples in an illustrative educational domain.

5 Conclusions

In this chapter we evaluate our results, which indicate that we reached the objective of usefully viewing process mining under a perspective based on planning as applied to a conceptual model specification, while, on the other hand, recognizing issues and shortcomings of our current achievement, and pointing out the most promising developments that our approach enables for future work.

5.1 Summary of the thesis

In this thesis we have presented an approach for the application of Process Mining tasks to domains with challenging characteristics. Our investigation is motivated by an attempt to combine goals and mechanisms contributed by the process mining research area, with those resulting from our treatment of information systems through an automated planning paradigm based on a conceptual modeling specification.

We discussed the related work in both areas in Chapter 2.

In Chapter 3 we discuss and describe our approach. We discuss how automated planning can account for unstructured processes in process mining in 3.1, and the characteristics of event logs in educational domains in 3.2. We defined the conceptual model that fundaments our approach as a Process Discovery task in 3.3, including how a conceptual model can be defined in relation to an available event log, supported by domain knowledge and the mining of typical plans for the Process Discovery of *de facto* models. In Section 3.4 we defined a plan-verification method and applied it to a Conformance Checking task, and also computed interest metrics of the domain for validation and analysis. In Section 3.5 we proposed and explored approaches for Model Enhancement.

In Chapter 4 we described the Library of Typical Plans for Process Mining that implements the methods discussed in Chapter 3, leveraging and expanding the prior art in the planning paradigm over a conceptual model specification, and detailed technical aspects of the implementations.

Throughout the thesis we discussed the methods and algorithms with examples in an academic program domain, which characterizes an unstructured process with the challenging characteristics that we concern ourselves with: *intertask dependencies, multiple dependencies, concurrent events, failing activities, repeated activities* and *knock-out structures*.

5.2 Evaluation

In this section we recapitulate the scope and objectives, determined in Chapter 1, and discuss how they relate to the methods and algorithms discussed throughout this thesis.

Recall our main hypothesis: that we are able to perform the typical Process Mining tasks in a domain with challenging characteristics via automated planning and a conceptual modeling paradigm. We posit that we have proven that the hypothesis is true to a reasonable degree. We have tackled the three main tasks, as defined in the seminal literature of the field.

In the process discovery task, we have discussed how a three-schemata conceptual model can be defined from the information contained in an event log. With the goal of simplicity of the resulting model in mind we have defined the concept of a case entity and formalized how the data in the event log can define a corresponding static schema. We only slightly extended the formulation in prior work to define value entities, useful for the constraint collection in the plan-recognition algorithms. We also defined how the information of the activities in the domain can be used to compose a dynamic schema. The operations, their effects and preconditions are implicitly defined from clauses describing the pre-requisites between activities. The case termination operations are defined from available case attributes. For the behavioral schema, we discussed how normative patterns can be formulated as goal-inference rules, but that will require domain knowledge input.

The most challenging aspect of the definition of a conceptual model for process mining is to define the model in an appropriate level of detail and complexity. This relates to the tradeoff between simplicity, fitness, precision and generalization of the model, and is a core concern of the process mining approach and other approaches relying on executable models. In typical process mining approaches, with graphical models like Petri Nets, the problem of determining and adequate level of detail for the model can be mitigated by the exploration of several techniques, especially in the process discovery task. Algorithms like the *HeuristicsMiner* can be parametrized for varying levels of detail, accounting for noise and infrequent behavior, in the resulting model. In our proposed approach we require domain knowledge to determine several aspects of the conceptual model.

On the other hand, still in the process discovery task, we find that the conceptual model easily represents the domains with the challenging characteristics outlined in our goals, and that the representation is adequate for the discovery of typical plans. These characteristics include the *intertask dependencies,* naturally captured by the pre-conditions of the operations - which also easily capture *multiple dependencies,* as the model allows for multiple sets of alternative preconditions. There’s also the characteristics of *failing activities*. We capture the failure (and success) of activities by modeling the effects of the operations conditionally, based on event attributes that can be used to determine whether the activity was a success. In the example of the educational domain, we leveraged the grade of the discipline attempt. By considering an event classifier in which figures the timestamp of the event in the trace – reflected by an additional argument in the operations – we also easily capture concurrent *operations*. In the same way, *repeated activities* are easily represented as different activity instances simply by having a different timestamp. Finally, the *knock-out structures* characteristic is captured by the modeling of alternative case termination operations.

As a result, given a conceptual model, tailored to represent the challenging aspects of the domain via operations and arguments in the operations, the automated planning algorithms are naturally capable of dealing with such characteristics. Only minor concerns had to be addressed in our approach, and they might not be universal to the automated planning techniques, as we discuss later. This strengthens our conclusion that the application of automated planning techniques based on a conceptual model for process mining is a feasible approach.

We also highlight that the generalizations of plans in the plan-recognition, as part of the process discovery task, capture richer detail of the behavior of cases without any downsides. For example, there is no downside to considering infrequent cases in the domain, which are a problem for some process discovery techniques for graphical models. In our example educational domain process, we find that students typically perform *unique* traces, with each student failing, skipping, re-ordering activities in a unique way – it is extremely rare for two students to perform the same disciplines in the same order, obtaining the same grades, etc. Yet, the plan-recognition algorithms are capable of finding meaningful representative plans of their behavior, given reasonably defined goal-inference rules. These characteristics are commonly associated to 'spaghetti processes' in graphical models, so called because of the complexity of the resulting representation.

The plan-recognition indeed also does not need to make special allowances to account for *ongoing* traces. Unfinished traces, in our approach, are easily represented by plots without case termination operations. The plan-mining will consider those plots just like the others when the effects of the case termination operation are not required to reach the situation and goal of a goal-inference rule. Thus, our approach also neatly deals with the *online* process discovery, as in *online process mining*.

A downside of the model in our approach is that is provides no way of visualizing the model. The visualization of the model is valuable both in the process analysis tasks as well as for visualizing the status of running cases. The latter feature would be especially important to our approach, since we deal with the online process mining. Furthermore, without visualization tools it is difficult to understand 'at a glance' the trace of a particular case of interest. While we can analyze cases of interest in detail – for example, by generating goal-inference rules to capture possible plans for recommendation purposes – the visualization of the cases, especially in comparison to a normative pattern, is a great way to provide inspection tools. With proper visualization tools a process stakeholder can intuitively follow a case, checking if it is performing a normative pattern or if it is discrepant. We could also leverage techniques for generating natural language from the planning operators, as we in explored in (DE LIMA, GOTTIN, *et al.*, 2017).

This issue relates to the conformance checking task. In our approach we find that the plan-verification method, leveraging the executable characteristic of the model, is capable of finding the discrepancy issues in detail, identifying not only the cases for which problems are found but also what are the activities that caused problems. The mechanism of 'allowing' failed operations for the sake of continuing with the replay checking is similar to those applied to graphical models. We additionally collect the failed preconditions of the nonconformant operations, which could support more intricate kinds of reporting.

In our analysis, we expected that the pre-requisite relationships between disciplines would ensure students perform conformant partial-orders in successful plans. However, there are a relevant number of discrepant operations in the students plots. We posit that this issue originates from pre-requisite sets of disciplines having *changed* in the syllabus definitions.

Our conformance checking approach also deals with the computation of interest metrics. We find that the reports comprising numeric scores satisfy the needs of comparing the relative relevance of rules and plans for analysis. Still, the visualization of all the metrics of all the plans could also benefit from visualization schemes. Another issue related to conformance checking is that he infrequent behavior, although not an issue for the plan-recognition approach, may generate skewed results for the fitness metrics. Finally, we did not explore plan-generation and plan-reachability techniques. We envision approaches, for example, in which traditional plan-generation techniques could be used to discover alternative conformant plans, upon discovering discrepant situations.

Mostly, this is due to our option of adopting simpler planning mechanisms than general purpose planners - recall from the discussion of our objectives that we did not intend to showcase all the representative power and features of planning approaches based on a conceptual model. Still, the adaptation of the methods and algorithms described in this thesis to leverage more power planning frameworks like the Interactive Plot Generator (CIARLINI, 1999) is one of the core concerns for our future developments. We strove to define methods and algorithms in such a way as to be easily adapted to other planning frameworks but have not performed in depth evaluation of the results of those efforts. We posit that we have succeeded, however, in keeping the Library of Typical Plans for Process Mining compatible with the BLIB (FURTADO e CIARLINI, 2001). The original BLIB was developed in the context of the IPG – hence, we trust that our resulting approach could be feasibly adapted to work with IPG, with no prohibiting issues. The only mechanism we defined that may conflict with the workings of more complex planners (IPG included) is the changes we enacted to the plan-recognition algorithm to account for concurrent operations ‘clobbering’ and ‘undoing’ effects of operations in the same timestamp. These same mechanisms could be further developed to enable the model enhancement by adding a richer *time perspective* to the model.

The model enhancement task is related to analyses tasks, and as such relies heavily on domain knowledge. We explored diverse possibilities, aiming to showcase practical applications in our domain – even if conservative in scope, due to the restrictions we faced. For example, we only had one additional case attribute with which to perform decision mining tasks. We highlight the importance of the model repair and model generalization approaches, since we started with a simpler model. The results we obtain are important to show that it is possible to iteratively amend the model to become more representative and reliable. The decision mining task is also important in that regard, as it enriches the model with characteristics of the cases. The representation of those additional characteristics in the conceptual model, and accounting for them in the planning techniques, is very straightforward.

One important aspect of the model enhancement task in our approach is the generation of new goal-inference rules. We determined that using only naïve transformations over pre-existing rules is insufficient and unfeasible in practice. While some goals generated by the straightforward lifting and combination of rules are interesting, we had to imbue the algorithms with mechanism to restrict the search-space to make them feasible. Those mechanism rely on domain knowledge to a point where it would debatably better for the user to write new rules for himself.

In any case, we identified rules of thumb for the combinations of rules that seem to be useful. One of them is to use the plot states reached by interesting cases to compose new goal-inference rules. Another one is that goal-inference rules in our process mining analysis typically deal with ‘positive behavior’ situations (e.g. students that performed well to a certain point) and goals (e.g. students will graduate) as well as ‘negative behavior’ situations (e.g. students that failed in the same discipline twice) and goals (e.g. will eventually drop or halt. For ‘positive’ situations and goals, we can generalize the rule by removing any failure terms (e.g. discipline failure from the situations). For ‘negative’ situations and goals we typically want to ensure the failure terms are part of the rules. The analysis of the relevance of these rules of thumb for other domains is still insipient, however, but indicates the possibility that the goal-inference rule generation mechanisms we proposed can be used at least for suggestion mechanisms, perhaps as part of a goal-inference rule editor tool.

Another major concern for future development of this work is the integration with tools for operational support. We are especially interested in methods for the conceptual design of information systems that lead to database implementation. By leveraging *database logs* – not only of domain-level events, but actual transactions in the database management system – we envision approaches where a static schema could be composed by applying *reverse engineering* (HEUSER, 2009, p. 119-144) to a running database. We particularly highlight the integration of the approach presented in this thesis to the IDB system (GOTTIN, DE LIMA e FURTADO, 2015). The IDB system implements an approach in which a conceptual model, in the same three-schemata format used in this thesis, is transitioned into a full DBMS environment. The functionalities provided allow the management of the database in a *story*-driven way, relying in (procedural versions of) operations of an automated planning domain. Even at the last stage the conceptual model specification is kept for allowing simulation, training, continuous testing, and many kinds of analysis based on statistical and artificial intelligence methods. Most importantly, the IDB system features a *temporal log* of the operations. Hence, the system not only extends the temporal database functionalities to deal with entities in the domain, but also deals with the *plots* of those entities over time. Finally, the IDB system features a prototypical implementation of storyboard features, similar to the ones presented in (DE LIMA, GOTTIN, *et al.*, 2017).

The IDB system relies on a more complete set of planning algorithms than the one used in our approach, however. The same considerations as for adapting our approach to the IPG planner likely apply. Furthermore, the choice of representation of discipline operations with the code of the discipline as the functor must be revisited – the abuse of notation will not carry straightforwardly to the IDB approach in which the definition of the operations in the dynamic schema has deeper consequences, including in the definition of the storage procedures that implement them in the DBMS.

As for the uses of the approach within IDB, we could use the simulation capabilities based on plan-generation that are part of the IDB systems, for example, for richer kinds of analysis and for obtaining prescriptive plans that have no support from previous cases. Our current approach can only provide recommendations for a student based on typical plans. These can be mined from a goal-inference rule composed from that particular students' current plot state, but that doesn't guarantee that a plan exists – for a plan to be recognized, prior cases must have been in similar situations before. Our methods for goal-inference rule generation ease those restrictions by generalizing the rules but can't provide the same exploratory power as a plan-generation technique.

Hence, the operational support provided by the integration of a tool like the IDB system and the approach presented in this thesis could configure an end-to-end approach to process mining linked to a database system. We could leverage the IDB log facilities to obtain *event logs*, likely with positive side effects to the quality of the data. That is, since the since the same system that performs the operations in the domain is charged with the logging, we can reasonably expect the log to be reliable and complete.

Furthermore, the same conceptual model specification could be used both to obtain a database instance in the IDB approach as well as to support the process discovery task in ours. The model discovery could alternatively focus on composing simplified, higher-level representations of the database model for specific kinds of analysis. This would help with dealing with the tradeoff between simplicity, fitness, precision and generalization, since a model for process mining could be created at the appropriate level of detail for specific analytics tasks, while the effective domain specification, in full complexity, is managed by a DBMS system in routine operational usage. The conformance checking (either of simplified or the actual database model) could, as typical, indicate opportunities for model enhancement which, in this context, relates to the '*meta-story*' of the evolution of the system during its lifetime.

5.3 Future Work

The main aspects for future developments of the present work are to generalize the approach – in formalization and implementation – towards other domains, and the integration of the presented approach within a system able to provide operational support. We deferred the integration of the approach into the IDB system, for example, for reasons discussed previously in Section 5.1. One aspect involved is the adoption of more complex planning mechanisms, as the one used in the IDB (or in the IPG) system. This too was anticipated in Section 5.1. In that section we also mentioned the need of providing the user with *inspection* mechanisms about the domain, possibly via visualization or plot narration mechanisms. We envision borrowing and extending features proper to *dramatization* approaches, developed in the context of systems wherein planning is used to compose plots over entertainment domains, as well as serious games domains, for which sophisticated dramatization is a major requirement.

Other topics of future work that we identified relate to the implementation of mechanisms for the management of the Library of Typical Plans for Process Mining. So far, we have focused our implementation on the building stage of the Library Index. The implementation could be extended to account for rebuilding processes, possibly leveraging prior work for re-structuring common plans in the Library Index, and also to enrich the dynamic schema with the discovered operations. We could also possibly bring in temporal model checking approaches for the conformance checking. Model checking is typically applicable to planning domains to verify hypotheses about complex properties of the system over time – accordingly, we could check the satisfiability of domain integrity rules expressed as temporal constraints over plots and plans.

Also related to conformance checking, we envision the collection of richer metrics of quality of plans. In the educational domain program, we already consider using the students' achievement scores as interest metrics of the model fitness. Besides that, we could consider the average grades obtained by students performing certain plans as indicative of the quality of those plans. This also relates to possible additional decision mining approaches, in which we could identify characteristics of the students (and/or their plots) that affect these metrics.

The current conformance checking replay mechanism additionally collects the failed preconditions encountered in discrepant operations in the plots but does not exploit them in interesting ways. We envision expanding the analysis of conformance checking discrepancies to consider the frequencies and patterns in these typically-failed preconditions. This information could lead to new kinds of reports – perhaps graphical reports, related to the visualization aspect discussed above – as also extend the decision mining approach. For example, we could identify the correlations between failing preconditions and characteristic classes of cases.

Finally, the Process Mining literature describes how data analytics tasks – classification models, sequence mining algorithms – applied over the event log can support the definitions and guide the scoping of the Process Mining tasks. Based on preliminary explorations we are particularly interested in *sequence mining* approaches as a way to identify candidate goal-inference rules.

6 References

ABELHA, P. et al. **Abelha, Paulo, et al. "A Nondeterministic Temporal Planning Model for Generating Narratives with Continuous Change in Interactive Storytelling**. AIIDE. [S.l.]: [s.n.]. 2013.

AGRAWAL, R.; GUNOPOULOS, D.; LEYMANN, F. **Mining process models from Workflow Logs.** Proceedings of the 6th International Conference on Extending Database Technology. [S.l.]: Springer-Verlag. 1998. p. 469-483.

AGRAWAL, R.; SRIKANT, R. **Fast algorithms for mining association rules**. Proceedings of the 20th International Conference on Very Large Databases (VLDB). Santiago, Chile: [s.n.]. 1994. p. 487-499.

ALBERTI, M. et al. **Agent Interaction in Abductive Logic Programming: the SCIFF proof-procedure**. DEIS. Bologna. 2006. (DEIS-LIA-06-001).

ALLEMAND, D. et al. **On the computational complexisty of hypothesis assembly**. Proceedings of the Tenth International Joint Conference on Artificial Intelligence. [S.l.]: [s.n.]. 1987.

APPICE, A. Towards mining the organizational structure of a dynamic event scenario. **Journal of Intelligent Information Systems**, v. 501, n. 1, p. 165-193, 2018. ISSN 10.1007/s10844-017-0451-x.

ATTIE, P. C. et al. **Specifiying and enforcing intertask dependencies**. 19th International Conference on Very Large Databases (VLDB). San Francisco, CA: Morgan Kaufmann. 1993. p. 134-145.

AWAD, A.; SMIRNOV, S.; WESKE, M. **Resolution of Compliance Violation in Business Process Models:** A Planning-Based Approach. OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Berlin: Springer. 2009. p. 6-23.

BATEMAN, J.; ZOCK, M. Natural language generation. **The Oxford Handbook on Computational Linguistics**, 2003.

BELLODI, E. R. F.; LAMMA, E. . **Probabilistic declarative process mining**. In International Conference on Knowledge Science, Engineering and Management. [S.l.]: [s.n.]. 2010. p. 293-303.

BOGARÍN, A.; CEREZO, R.; ROMERO, C. A survey on educational process mining. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, v. 8, n. 1, 2018.

BOSE, R. P. J. C.; VERBEEK, E. H. M. W.; VAN DER AALST, W. M. P. **Discovering Hierarchical Process Models Using ProM**. Forum the Conference on Advanced Information Systems Enginieering (CAisE). [S.l.]: [s.n.]. 2011.

BRIN, S. et al. Dynamic itemset counting and implication rules for market basket data. **Acm Sigmod Record**, v. 26, n. 2, p. 255-264, 1997.

BROWNE, M. C.; CLARKE, E. M.; GRÜMBERG, O. Characterizing finite Kripke structures in propositional temporal logic. **Theoretical Computer Science**, v. 59, n. 1-2, p. 115-131, 1988.

CASANOVA, M. A. et al. Three Decades of Research on Database Design at PUC-Rio. **Journal of Information and Data Management**, v. 3, n. 1, p. 19-34, 2012.

CHESANI, F. et al. Exploiting inductive logic programming techniques for declarative process mining. **Transactions on Petri Nets and Other Models of Concurrency**, p. 278-295, 2009.

CHRISTIANSEN, H. Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules. **Journal of Applied Logic**, v. 7, n. 3, p. 341-362, 2009.

CHRISTIANSEN, H.; DAHL, V. **HYPROLOG:** A new logic programming language with assumptions and abduction. International Conference on Logic Programming. [S.l.]: [s.n.]. 2005.

CIARLINI, A. E. M. **Geração interativa de enredos.** PUC-Rio. Rio de Janeiro. 1999.

CIARLINI, A. E. M. et al. Event Relations in Plan-Based Plot Composition. **Computers in Entertainment (CIE)**, v. 7, 2007. ISSN DOI: http://doi.acm.org/10.1145/1658866.1658874.

CIARLINI, A. E. M. et al. Modeling interactive storytelling genres as application domains. **Journal of Intelligent Information Systems** , v. 35, n. 3, p. 347-381, 2010.

CIARLINI, A. E. M.; FURTADO, A. L. Understanding and Simulating Narratives in the Context of Information Systems. **ER**, p. 291-306, 2002.

CODOGNET, P.; DIAZ, D. Compiling constraints in clp(FD). **The Journal of Logic Programming**, p. 185-226, 1996.

CONSOLE, L.; DUPRÉ, D. T.; TORASSO, P. On the relationship between abduction and deduction. **Journal of Logic and Computation**, v. 1, 1991.

CONSOLE, L.; DUPRE, T.; TORASSO, P. **Causes for events:** their computations and applications. Eight International Conference on Automated Deduction. [S.l.]: [s.n.]. 1986. p. 608-621.

CRESSWELL, S. N.; MCCLUSKEY, T. L.; WEST, M. M. Acquiring planning domain models using LOCM. **The Knowledge Engineering Review**, v. 28, n. 2, p. 195-213, 2013.

DA SILVA, F. A. G.; CIARLINI, A. E. M.; SIQUEIRA, S. W. M. **Nondeterministic Planning for Generating Interactive Plots**. In: Kuri-Morales A., Simari G.R. (eds) Advances in Artificial Intelligence – IBERAMIA 2010. IBERAMIA 2010. Lecture Notes in Computer Science. Berlin: [s.n.]. 2010.

DAHL, V.; TARAU, P. **Assumptive Logic Programming**. Argentine Symposium on Artificial Intelligence. [S.l.]: [s.n.]. 2004.

DE KLEER, J. An assumption-based TMS. **Artificial Intelligence**, p. 127-162, 1986.

DE LEONI, M.; MARRELLA, A. Aligning real process executions and prescriptive process models through automated planning. **Expert Systems with Applications,** v. 82, n. 162-183, 2017.

DE LIMA, E. S. **Video-Based Interactive Storytelling**. PUC-Rio. [S.l.]. 2014.

DE LIMA, E. S. et al. **Network Traversal as an Aid to Plot Analysis and Composition**. Proceedings of SBGames 2017. [S.l.]: [s.n.]. 2017. p. 418-427.

DENECKER, M. . D. S. D. SLDNFA: an abductive procedure for abductive logic programming. **Logic Programming** , v. 34, n. 2, p. 111-167, 1998.

DENECKER, M.; MISSIAEN, L.; BRUYNOOGHE, M. Temporal Reasoning with Abductive Event Calculus. **ECAI**, 1992.

DÓRIA, T. R.; CIARLINI, E. M.; ANDREATTA, A. **A nondeterministic model for controlling the dramatization of interactive stories**. Proceedings of the 2Nd ACM International Workshop on Story Representation, Mechanism and Context. Vancouver, British Columbia, Canada: ACM. 2008. p. 21-26.

DOURISH, P. et al. **Freeflow:** Mediating Between Representation and Action in Workflow Systems. New York: In Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '96). 1996.

EITER, T.; GOTTLOB, G.; LEONE, N. Semantics and complexity of abduction from defautl theories. **Journal of Artificial Intelligence**, v. 90, p. 177-223, 1997.

ESHGHI, K.; KOWALSKI, R. **Abduction as deduction**. Imperial College. London. 1988.

ESHGHI, K.; KOWALSKI, R. **Abduction compared with negation by failure**. Imperial College. London. 1989.

EXPLANATION and prediction: an architecture for default and abductive reasoning. **Computational Intelligence**, v. 52, p. 97-110, 1989.

FANN, K. T. **Peirce's theory of abduction**. The Hague: Martinus Nihjoff, 1970.

FERNANDES, A. et al. Adding flexibility to workflows through incremental planning. **Innovations in Systems and Software Engineering**, v. 3, n. 4, p. 291-302, 2007.

FERREIRA, H.; FERREIRA, D. An integrated life cycle for workflow management based on learning and planning. **International Journal of Cooperative Information Systems**, v. 15, n. 4, p. 485-505, 2006.

FIKES, R. E.; NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. **Artificial intelligence**, v. 2, n. 3-4, p. 189-208, 1971.

FLIEDL, G.; KOP, C.; VÖHRINGER, J. Guideline based evaluation and verbalization of OWL class and property labels. **Data & Knowledge Engineering**, v. 69, n. 4, p. 331-342, 2010.

FORD, M. **An Abductive reasoning system in Java**. [S.l.]. 2012.

FRÜHWIRTH, T. Theory and practice of constraint handling rules. **The Journal of Logic Programming**, p. 95-138, 1998.

FURTADO, A. et al. Applying Analogy to Schema Generation. **iSys-Revista Brasileira de Sistemas de Informação**, v. 1, n. 1, 2008.

FURTADO, A. L. Analogy by generalization - and the quest of the grail. **ACM SIGPLAN Notices**, v. 27, n. 1, January 1992.

FURTADO, A. L. Narratives and temporal databases: an interdisciplinary perspective. **Conceptual modeling: historical perspectives and future directions**, 1999.

FURTADO, A. L. Narratives and Temporal Databases: An Interdisciplinary Perspective. In: CHEN, P. P., et al. **Conceptual Modeling:** Current Issues and Future Directions. [S.l.]: Springer, 1999.

FURTADO, A. L. et al. **Plot mining as an aid to characterization and planning**. PUC-Rio. Rio de Janeiro. 2007. (MCC 07/07).

FURTADO, A. L. et al. **Plot Mining as an Aid to Characterization and Planning**. Departamento de Informática - PUC Rio. Rio de Janeiro, p. 22. 2007. (ISSN 0103-9741).

FURTADO, A. L. et al. **Analysis and Reuse of Plots Using Similarity and Analogy**. Proc. 27th International Conference on Conceptual Modeling. Barcelona: [s.n.]. 2008. p. 355-368.

FURTADO, A. L.; CASANOVA, M. A. **Plan and Schedule Generation over Temporal Databases**. Prc. 9th Internation Conference on Entity-Relationship Approach (ER 90). [S.l.]: [s.n.]. 1990. p. 235-248.

FURTADO, A. L.; CASANOVA, M. A.; BARBOSA, S. D. J. A Semiotic Approach to Conceptual Modelling. **Internation Conference on Conceptual Modeling**, Cham, 2014.

FURTADO, A. L.; CIARLINI, A. E. M. **Plots of Narratives over Temporal Databases**. Databases and Expert Systems Applications Workshop. Toulouse, France: [s.n.]. 1997.

FURTADO, A. L.; CIARLINI, A. E. M. **Generating Narratives from plots using schema information**. International Conference on Application of Natural Language to Information Systems. Berlin: Springer. 2000.

FURTADO, A. L.; CIARLINI, A. E. M. The plan recognition / plan generation paradigm. **Information Systems Engineering - State of the Art and Research Themes**, 2000.

FURTADO, A. L.; CIARLINI, A. E. M. **Constructing libraries of typical plans**. International Conference on Advanced Information Systems Engineering. [S.l.]: [s.n.]. 2001.

FURTADO, A.; CIARLINI, A. **Aiding the Construction of Libraries of Typical Plans**. PUC Rio. [S.l.]. 2000. (ISSN 0103-9741).

GABBAY, D. M.; WOODS, J. **A Practical Logic of Cognitive Systems:** The reach of Abdcution: Insight and Trial. [S.l.]: Elsevier, v. 2, 2005.

GATT, A.; REITER, E. **SimpleNLG:** A realisation engine for practical applications. Proceedings of the 12th European Workshop on Natural Language Generation. [S.l.]: [s.n.]. 2009. p. 90-93.

GHALLAB, M.; NAU, D.; TRAVERSO, P. **Automated Planning Theory and Practice**. San Francisco, CA: Morgan Kaufmann Publishers Inc, 2004.

GHALLAB, M.; NAU, D.; TRAVERSO, P. **Automated planning and acting**. Authors' manuscript. ed. [S.l.]: Cambridge University Press, 2016.

GLANCE, N.; PAGANI, D.; PARESCHI, R. **Generalised Process Structure Grammars (GPSG) for Flexible Representations of Work**. New York: In Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW96). 1996.

GOEDERTIER, S. **Declarative Techniques for Modeling and Mining Business Processes**. Katholieke Universiteit Leuven. [S.l.]. 2008.

GONÇALVES, E. M. N.; BITTENCOURT, G. **A Planning-Based Knowledge Acquisition Methodology**. LAPTEC. [S.l.], p. 105-112. 2005.

GONZÁLEZ-FERRER, A.; FERNÁNDEZ-OLIVARES, J.; CASTILLO, L. From business process models to hierarchical task network planning domains. **THe Knowledge Engineering Review**, v. 28, n. 2, p. 175-193, 2013.

GORDON, A. S. **Commonsense Interpretation of Triangle Behavior**. AAAI Conference on Artificial Intelligence. [S.l.]: [s.n.]. 2016.

GOTTIN, V. M. **Verificação Abstrata de Propriedades Dramáticas Contínuas em Eventos Não-Determinísticos**. UNIRIO. [S.l.]. 2013.

GOTTIN, V. M. et al. **A Story-Based Approach to Information Systems**. [S.l.]. 2015. (MCC05/15).

GOTTIN, V. M. et al. **An Analysis of Degree Curricula through Mining Student Records**. Advanced Learning Technologies (ICALT), 2017 IEEE 17th International Conference. [S.l.]: [s.n.]. 2017. p. 276-280.

GOTTIN, V. M..; JIMENEZ, H. G. **Academic Analytics at PUC-Rio**. PUC-Rio. [S.l.]. 2017.

GOTTIN, V. M.; DE LIMA, E. S.; FURTADO, A. L. **Applying Digital Storytelling to Information System Domains**. Proceedings of the XIV Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2015). [S.l.]: [s.n.]. 2015.

GOTTIN, V. M.; JIMENEZ, H. G. Information Visualization. **Gottin V. M. academic page**, 2017. Disponivel em: <http://www.inf.puc-rio.br/~vgottin/projects/inf2691.html>. Acesso em: 14 November 2019.

HALLER, S. M. **A model for cooperative interactive plan explanation**. Artificial Intelligence for Applications, 1994., Proceedings of the Tenth Conference on. [S.l.]: [s.n.]. 1994.

HAN, J.; PEI, J.; YIN, Y. Mining frequent patterns without candidate generation, v. 29, n. 2, p. 1-12, 2000.

HEINL, P. et al. **A Comprehensive Approach to Flexibility in Workflow Management Systems**. New York, NY, USA. 1999.

HEINRICH, B.; SCHÖN, D. **Automated planning of context-aware process models.** University of Regensburg. Regensburg. 2015.

HEUSER, C. A. **Projeto de banco de dados:** Volume 4 da Série Livros didáticos informática UFRGS. [S.l.]: Bookman Editora, v. 4, 2009.

HICKMOTT, S. **Concurrent planning using Petri net unfoldings**. [S.l.]: [s.n.]. 2006. p. International Conference on Automated Planning and Scheduling-ICAPS.

HNICH, B.; KIZILTAN, Z.; WALSH, T. **Modelling a balanced academic curriculum problem.** Proceedings of CP-AI-OR-2002. [S.l.]: [s.n.]. 2002.

HORNIK, K.; GRÜN, B.; HAHSLER, M. arules-A computational environment for mining association rules and frequent item sets. **Journal of Statistical Software**, v. 14, n. 15, p. 1-25, 2005.

HOUSTMA, M.; SWAMI, A. **Set-oriented mining for association rules in relational databases**. In Data Engineering, 1995. Proceedings of the Eleventh International Conference. [S.l.]: IEEE. 1995. p. 25-33.

JAFAR, J. et al. ACM Transactions on Programming language and system, p. 339-395, 1992.

JIMÉNEZ, H. G. **Applying Process Mining to the Academic Adminstration Domain**. PUC-Rio. [S.l.]. 2017.

JUHÁSOVA, A. et al. **How to model curricula and learnflows by Petri nets-a survey**. In Emerging eLearning Technologies and Applications (ICETA). [S.l.]: [s.n.]. 2016.

KAKAS, A. C.; KOWALSKI, R. A.; TONI, F. The role of abduction in logic programming. **Handbook of logic in artificial intelligence and logic programming**, v. 5, p. 235-324, 1998.

KAKAS, A. C.; KOWALSKI, R. A.; TONI, F. The role of abduction in logic programming. **Handbook of logic in artificial intelligence and logic programming**, 5, 1998. 235-324.

KAKAS, A. C.; KOWALSKI, R.; TONI, F. Abductive Logic Programming. **Journal of Logic and Computation**, v. 2, n. 6, p. 719-770, 1992.

KAKAS, A. C.; MICHAEL, A.; MOURLAS, C. ACLP: Abductive constraint logic programming. **Journal of Logic Programming**, v. 44, n. 1, p. 129-177, 2000.

KAKAS, A. C.; NUFFELEN, B. V.; DENECKER, M. **A-System:** Problem Solving through abduction. International Conference on Logic Programming and Nonmonotonic Reasoning. Berlin: [s.n.]. 2001.

KAKAS, A. C.; PAPADOPOULOS, G. A. **Parallel Abduction in Logic Programming**. 1st International Symposium on Parallel Symbolic Computation (PASCO). Linz, Austria: [s.n.]. 1994. p. 214-224.

KATZOURIS, N.; ARTIKIS, A.; PALIOURAS, G. **ILED**. [S.l.]. 2014.

KAUTZ, H. A. **A formal theory of plan-recognition and its implementation**. San Mateo: Morgan Kaufmann, 1991.

LEEMANS, M.; VAN DER AALST, W. M. **Discovery of frequent episodes in event logs**. In International Symposium on Data-Driven Process Discovery and Analysis. [S.l.]: Springer. 2014. p. 1-31.

LEEMANS, S. J.; FAHLAND, D.; VAN DER AALST, W. M. **Discovering block-structured process models from event logs containing infrequent behaviour**. In International conference on business process management. [S.l.]: Springer. 2013. p. 66-78.

LEVESQUE, H. L. Foundations of a functional approach to knowledge representaiton. **Artificial Intelligence**, v. 23, p. 155-212, 1984.

LEVESQUE, H. L. **A knowledge-level account of abduction**. Processings of IJCAI 89. [S.l.]: [s.n.]. 1989. p. 1061-1067.

LLERA, A. A. **Seeking Explanations:** Abduction In Logic Philosophy Of Science And Artificial Intelligence. Amsterdam: Universeteir van Amsterdam, 1997.

LY, L. T. et al. **Data transformation and semantic log purging for process mining**. In International Conference on Advanced Information Systems Engineering. [S.l.]: Springer. 2012. p. 238-253.

MA, J. **User Guide of the SICStus abduction module: abductive logic programming for Prolog**. [S.l.]. 2011.

MANHARDT, F. et al. **Data-driven process discovery-revealing conditional infrequent behavior from event logs.** 2009 International Conference on Information, Process, and Knowledge Management. [S.l.]: IEEE. 2009. p. 545-560.

MANNILA, H.; TOIVONEN, H.; VERKAMO, A. I. Discovery of frequent episodes in event sequences. **Data Mining and Knowledge Discovery**, v. 1, n. 3, p. 259-289, 1997.

MARRIOT, K.; STUCKEY, P. J. **Introduction to Constraing Logic Programming**. [S.l.]: [s.n.], 1998.

MCDERMOTT, D. **The formal semantics of processes in PDDL**. Proc. ICAPS Workshop on PDDL. [S.l.]: [s.n.]. 2003. p. 101-155.

MCDERMOTT, D. et al. **PDDL-the planning domain definition language**. [S.l.]. 1998.

MCILRAITH, S. A. **Logic-based abductive inference**. Knowledge Systems Laboratory. [S.l.]. 1998.

MOONEY, R. J. Learning Plan Schemata from Observation: Explanation-Based Learning for Plan Recognition. **Cognitive Science**, v. 14, n. 4, p. 483-509, 1990.

MOONEY, R. J. Integrating Abduction and Induction to Machine Learning. [S.l.]: Kluwer Academic, 2000. p. 181-191.

PAAVOLA, S. Abduction as a logic and methodology of discovery: the importance of strategies. **Foundations of Science**, v. 9, n. 3, p. 267-283, 2004.

PAPAMARKOS, G.; POULOVASSILIS, A.; WOOD, P. T. **Event-condition-action rule languages for the semantic web**. Proceedings of the First International Conference on Semantic Web Databases. [S.l.]: [s.n.]. 2007. p. 294-312.

PEARL, J. **Probabilistic Reasoning in Intelligent Systems**. [S.l.]: Morgan Kaufmann, 1998.

PECHENIZKIY, M. et al. CurriM: curriculum mining. **Educational Data Mining**, 2012.

PENG, Y.; REGGIA, J. **Abductive Inference Models for Diagnostic Problem-Solving**. [S.l.]: Springer-Verlag, 1990.

PENG, Y.; REGGIA, J. **Abductive inference models for Diagnostic Problem-Solving**. [S.l.]: Springer-Verlag, 1990.

PESIC, M. **Constraint-based workflow management systems: shifting control to users**. Technische Universiteit Eindhoven. Eindhoven. 2008.

PICCININI, H. et al. **Verbalization of RDF tiples with applications**. ISWC-Outrageous Ideas Track. [S.l.]: [s.n.]. 2011.

PICHLER, P.; ET AL. **Imperative versus declarative process modeling languages:** an empirical investigation. International Conference on Business Process Management. Berlin: [s.n.]. 2011.

POOLE, D. A logical framework for default reasoning. **Artificial Intelligence**, v. 36, n. 1, p. 27-27, 1987.

POOLE, D. What the Lottery Paradox tells us about Default Reasoning. **Knowledge Representation**, 1989.

POOLE, D. **Compiling a Default Reasoning System into Prolog**. New Generation Computing. [S.l.]: [s.n.]. 1991. p. 3-38.

POOLE, D. **AILog User Manual Version 2.3**. [S.l.]. 2008.

POOLE, D. **Local Users Guide to Theorist**. [S.l.].

POOLE, D.; GOEBEL, R. G.; ALELIUNAS, R. Theorist: a logical reasoning system for defaults and diagnosis. In: CERCONE, N.; MCCALLA, G. **The Knowledge Frontier:** Essays in the Representation of Knowledge. [S.l.]: Springer-Verlag, 1987. p. 331-352.

POOLE, D.; LYNTON, A. K.; GOEBEL, R. **Computational Intelligence:** a logical approach. [S.l.]: Oxford University Press, 1998.

POOLE, D.; MACKWORTH, A.; GOEBEL, R. **Computational Intelligence:** A Logical Approach. [S.l.]: Oxford University Press, 1998.

POPLE, H. E. **On the mechanization of abductive logic**. IJCAI. [S.l.]: [s.n.]. 1973.

RAY, O. Nonmonotonic Abductive Inductive Learning. **Journal of Applied Logic**, p. 329-340, 2009.

REGGIA, J. Diagnostic experts systems based on a set-covering model. **International Journal of Man-Machine Studies**, v. 19, n. 5, p. 437-460, 1983.

REITER, R. A logic for default reasoning. **Artificial Intelligence**, p. 81-132, 1980.

RODRIGUES, P. S. L. et al. **An Expressive Talking Head Narrator for an Interactive Storytelling System**. PUC-Rio. [S.l.]. 2015. (MCC 15/05).

RODRIGUEZ, C. et al. **Eventifier:** Extracting process execution logs from operational databases. In Demonstration Track of BPM Conference, CEUR-WS. [S.l.]: [s.n.]. 2012. p. 17-22.

ROSSI, F.; VAN BEEK, P.; WALSH, T. **Handbook of Constraint Programming**. [S.l.]: Elsevier, 2006.

SAVASERE, A.; OMIECINSKI, E. R.; NAVATHE, S. B. **An efficient algorithm for mining association rules in large databases.** Georgia Institute of Technology. [S.l.]. 1995.

SCHONENBERG, M. H. et al. **Taxonomy of process flexibility**. [S.l.]. 2007.

SCHRIJVERS, T.; DEMOEN, B. **The K.U.Leuven CHR System: Implementation and Application**. [S.l.]. 2004.

SNOWDON, A. et al. On the Architecture and Form of Flexible Process Support. **Software Process Improvement and Practice**, v. 12, p. 21-34, 2007.

SOWA, J. **Knowledge Representation:** logical, philosophial, and computational foundations. [S.l.]: [s.n.], 1999.

STRYCZEK, R. Petri net-based knowledge acquisition framework for CAPP. **Advances in Manufacturing Science and Technology**, v. 32, n. 1, p. 21-38, 2008.

SURIADI, S. et al. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. **Information Systems**, v. 64, p. 132-150, 2017.

TRCKA, N.; PECHENIZKIY, M. **From local patterns to global models:** Towards domain driven educational process mining. In Intelligent Systems Design and Applications, 2009. [S.l.]: [s.n.]. 2009. p. 1114-1119.

VAN DER AALST, W. M. Re-engineering knock-out processes. **Decision Support Systems**, v. 30, n. 4, p. 451-468, 2001.

VAN DER AALST, W. M. **Process Mining:** Discovery, Conformance and Enhancement of Business Processes. [S.l.]: Springer, Heidelberg, Dordrecht, London et. al, 2011.

VAN DER AALST, W. M. **Extracting event data from databases to unleash process mining**. In BPM-Driving innovation in a digital world. [S.l.]: [s.n.]. 2015. p. 105-128.

VAN DER AALST, W. M. P. Business alignment: using process mining as a tool for Delta analysis and conformance testing. **Requirements Engineering**, v. 10, n. 3, p. 198-211, 2005.

VAN DER AALST, W. M. P. **Process Mining - Discovery, Conformance and Enhancement of Business Processes**. [S.l.]: [s.n.], 2011.

VAN DER AALST, W. M. P. et al. **Process mining manifesto**. Berlin: Springer. 2011.

VAN DER AALST, W. M. P.; WEIJTERS, A. J. M. M. Process mining: a research agenda. **Computers in Industry**, v. 53, n. 3, p. 231-244, 2004.

VAN DER AALST, W.; DE BEER, H. T.; VAN DONGEN, B. F. **Process mining and verification of properties:** An approach based on temporal logic. "OTM Confederated International Conferences" On the Move to Meaningful Internet Systems. [S.l.]: [s.n.]. 2005.

VAN DONGEN, F. et al. **The ProM framework:** A new era in process mining tool support. International Conference on Application and Theory of Petri Nets. Berlin: Heidelberg. 2005. p. 444-454.

VELOSO, M. M.; CARBONELL, J. G. Derivatioal Analogy in PRODIGY: Automating Case Acquisition, Storage and Utilization. In: ______ **Case-Based Learning**. [S.l.]: [s.n.], 1993. p. 249-278.

VERBEEK, H. M. W. et al. **Prom 6:** The process mining toolkit. Proc. of BPM Demonstration Track. [S.l.]: [s.n.]. 2010. p. 34-39.

VERBEEK, H. M.; BUIJS, J. C.; VAN DONGEN, B. G. . V. D. A. W. M. **XES, XESAME and PROM 6**. International Conference on Advanced Information Systems Engineering. [S.l.]: [s.n.]. 2010.

WAINER, J.; DE LIMA BEZERRA, F. **Constraint-based flexible workflows**. [S.l.]: In Proceedings of the 9th International Workshop on Groupware: Design, Implementation, and Use (CRIWG 2003). 2003.

WANG, R.; ZAÏANE, O. R. **Discovering Process in Curriculum Data to Provide Recommendation**. [S.l.], p. 580-581. 2015.

WEIJTERS, A. J. M. M.; RIBEIRO, J. T. S. **Flexible heuristics miner (FHM)**. IEEE symposium on computational intelligence and data mining (CIDM). [S.l.]: [s.n.]. 2011. p. 310-317.

WEIJTERS, A. J. M. M.; VAN DER AALST, W. M.; DE MEDEIROS, A. A. **Process mining with the heuristics miner-algorithm**. Technische Universiteit Eindhoven. [S.l.], p. 1-34. 2006.

YANG, Q.; TENENBERG, J.; WOODS, S. On the Implementation and Evaluation of Abtweak. **Computational Intelligence Journal**, v. 12, n. 2, p. 295-318, 1996.

ZAKI, M. J. SPADE: an efficient algorithm for mining frequent sequences. **Machine Learning**, v. 42, n. 1, p. 31-60, 2001.

ZAKI, M. J. et al. **New algorithms for fast discovery of association rules**. KDD. [S.l.]: [s.n.]. 1997. p. 283-286.

ZAKI, M. J.; MEIRA JR., W. **Data mining and analysis:** fundamental concepts and algorithms. [S.l.]: Cambridge University Press, 2014.

ZAKI, M.; LESH, N.; OGIHARA, M. Planmine: Predicting plan failures using sequence mining. **Artificial Intelligence Review**, v. 14, n. 6, p. 421-446, 2000.

ZIMMERMANN, A. Colored petri nets. In: ZIMMERMANN, A. **Stochastic Discrete Event Systems:** Modeling, Evaluation, Applications. [S.l.]: [s.n.], 2008. p. 99-124.

7 Appendix - Educational Domain Definitions

In this thesis we perform the process mining tasks over an academic program example. The description of the original data sources, from which the event logs were obtained, and the explanation of preprocessing steps performed are available at (GOTTIN e JIMENEZ, 2017)[3]. In the following we describe that domain, discussing the terminology adopted; and also discuss the challenging characteristics of the domain that motivate its use as an example for process mining.

Terminology

Due to the varied terminologies used by academic systems we provide a list of the usual synonyms, when applicable, for each defined term. These are given in footnotes in this Appendix.

The ***academic program***[4] is an undergraduate educational program at a higher learning institution - PUC-Rio. Each program is designed with general and specific objectives in mind, defined in accordance with regulations processes, both internal to PUC-Rio as well as pertaining to the Brazilian government body that regulates higher learning institutions. Each program is periodically updated, also in accordance to those regulations.

The program dictates the activities and requirements that the student must perform and achieve in order to obtain a degree. These requirements determine an expected number of activity-hours the student must complete, as well a minimum, expected and maximum number of ***academic terms***[5] for performing those activities.

[3] The report and additional material are available at http://www.inf.puc-rio.br/~vgottin/projects/inf2391.html (accessed 2019, November 14th).

[4] In the terminology adopted at PUC-Rio the academic program is a *course*. We avoid the usage of the term *course* because it is also typically used, in other higher learning institutions, as the term for what we call a *discipline*.

[5] In the terminology adopted at PUC-Rio the generic term for *academic term* is a *'period'* ("período"). We avoid using that term because it is frequently used, in other higher learning institutions, to refer to the scheduling of the classes, lectures or sessions for a discipline, to distinguish between morning, afternoon and evening.

In the case of the academic programs at PUC-Rio the academic terms correspond to *academic semesters*. Finally, the activities required for graduation are expressed in terms of ***credits*** that the student must obtain in each of the categories of activities. The activities mainly comprise ***disciplines***[6] (but also other, *complementary activities* – e.g. participation scientific conferences, internships, etc.). Certain disciplines are considered 'basic', part of an initial shared common sequence of disciplines for several programs (e.g. "MAT 1157", "Single Variable Calculus A" is part of the basic set of disciplines for all Engineering programs). In our analysis, we focus on a single program – Computer Science. Hence, we don't consider this aspect in the definition of disciplines.

Students are identified in the program by their ***enrollment*** number, a unique registration number that states the year and academic term of first enrollment. For practical purposes, we anonymized the dataset, as discussed later on in this Appendix: that is, we refer to students by a unique anonymized identification number. Students enroll in disciplines in each semester and have their performances evaluated, obtaining a record of ***success*** or ***failure***[7]. Students may be evaluated in many ways – there are ten distinct evaluation categories defined at PUC-Rio, defining number, frequency and relative weight of tests, grading schemes, etc. Typically, the ***grade***[8] is a number that summarizes the performance of the student. The sequence of disciplines performed by the student is that student's *trace*, in process mining terminology (and *plot*, in ours). At PUC-Rio the information of the activities performed by the student is part of his academic record[9].

As stated, the activities are linked to *credits*. Each discipline confers a number of *credits* to the students who are successful, determined according to the discipline's *syllabus* in the definition of the academic program. Besides the credits,

[6] Disciplines are also referred to, in other works related to educational domains, as *courses*, *classes* or *lectures*.

[7] In the terminology adopted at PUC-Rio the student is either *approved* ("aprovado") or *rejected* ("reprovado"). We choose *success* and *failure* as these terms are closer to the terminology that seems to be used in formal contexts in English. A discipline *failure* is colloquially referred to as *flunking*.

[8] The *grade* ("grau") obtained by the student at PUC-Rio is typically expressed as a number between 0 and 10, with a single digit for the fractional part, e.g.: 8.7, 1.0, 2.6, etc. To simplify the representation, we adopt a range of integer grades between 0 and a 100, and no fractional part.

[9] We mainly use *trace* and *plot* throughout the thesis, avoiding the term 'academic record' ("histórico"). In other works and systems the sequence of disciplines and activities performed by a student is called her *career* or *transcript*.

the syllabus states the discipline's (human-readable) name, topics (the subjects covered by the lectures, classes or laboratory sessions), the required and suggested bibliography, the evaluation category, the ***pre-requisite*** disciplines and the identifying ***code***.

The *code* of a discipline at PUC-Rio is comprised of a three-letter abbreviation of the department that offers that discipline, plus a four-digit unique number. E.g. 'INF' is department of informatics; 'MAT' is the mathematics department. It is common for the academic programs pertaining to a department to have several disciplines of other departments as required activities. For example, there are several disciplines from the Engineering and Mathematics departments for the Computer Science course, in the department of informatics. There is also meaning in the composition of the four-digit number (the first digit '1' denotes an undergraduate-level discipline, for example), but we ignore that in the examples of this thesis.

The *pre-requisite* of a discipline are given as sets of alternative pre-requisites. For example, discipline INF1019 requires that the student have successfully completed *either* INF1612 *or* INF1018. Some disciplines have as a pre-requisite requirement a minimum number of *credits*. For example, discipline INF1014 requires the student have obtained 120 credits; discipline INF1608 requires 100 credits *and* that the student have successfully completed INF1001.

It is not permitted for a student to re-enroll in a discipline in which she was already successful. The student is allowed, and oftentimes required, to re-enroll in disciplines in which she has failed. There are also rules governing a minimum and maximum amount of disciplines a student can perform each semester (expressed as a number of maximum *credits*, e.g. 30 credits in disciplines in a term for the Computer Science program).

The academic program also determines the ***recommended term*** (or ***recommended semester***) for each discipline that is part of the required activities. The set of required disciplines, along with the *recommended semesters* is what we call the academic program's ***curriculum***. Hence, the curriculum is a normative pattern – a *'de juris model*', the suggested set and sequence of activities for students to follow in the program.

The student that obtains the required number of credits (that is, successfully completes the required activities) within the maximum number of academic terms allowed ***graduates***. That is, he successfully completes the academic program. The students that exceed that maximum number of academic terms allowed without fulfilling all the requirements are *dismissed*[10] from the program. There are several other reasons for which a student can interrupt the academic program, either temporarily or permanently. E.g., if a student fails the same discipline 5 times, he is forcibly dismissed from the program as well. Other reasons yet relate to internal management issues at PUC-Rio and are unrelated to the student's academic performance (e.g. the student can be dismissed from the program as punishment for administrative infractions).

In the record systems used at PUC-Rio the student's *status* in an academic term can be one of 18 alternatives, several of which relate to each one of: graduated students, students currently enrolled in the program, students temporarily suspended from program, and students permanently dismissed from the program. There are cases in which a student *re-enrolls* in the academic program by performing (and succeeding) in the admission process. These students receive a new enrollment number, but the successful (and unsuccessful) disciplines are carried over from the previous enrollment to the new one.

In our approach we consider all the students that are permanently dismissed from the program as ***dropped out*** students. We also ignore complex cases dealing with transferred students – to or from other higher learning institutions. Hence, we initially consider the three possible 'status' groups: *graduated* students (successful traces), *enrolled* students (that is, partial traces, still performing the process) and *dropped out* students (unsuccessful traces). As part of the model enhancement task we additionally consider students that are *halted* (also partial traces).

Academic program as an educational process

Recall from Section 2.1 our discussion on van der Aalst's (2011) Process Mining definitions of *process*, *case*, *activity* and *event*. Recall also the

[10] The terminology used at PUC-Rio ("jubilado") also corresponds to the 'sent down' term, used in British English.

correspondence (Section 3.2) between those terms and the academic program: in our educational domain example we consider an academic program as the process, with each student's career throughout the program configuring a case. The activities in the process are the discipline, and events may refer to the approval (being assigned a passing grade) or the failing (a failing grade) of a *discipline*.

We now explore additional typical restrictions over the log discussed by van der Aalst (2011, p. 97-103) in the context of our educational domain. This discussion adds to the discussion in Section 3.2.

- "One event log corresponds to one process":
  Our event log corresponds to a single degree program.
- "Only relevant events in the data":
  Only events relating to enrollment, enrollment cancellation, approval, passing, failing or abandoning courses are present in the log.
- "Specify a process as a collection of activities such that the lifecycle of a single instance is described":
  The degree program is specified as a collection of courses such that the career of a single student is described.
- "*case id* and *activity* columns (…) bare minimum for process mining"
  Student id and discipline code must figure in each entry in the log.
- "Events within a case are ordered":
  Events within a student's plot are ordered. However, since we deal with concurrent events, there are several alternative plots representing the same orderings.

Additionally, when considering the concept of a classifier (VAN DER AALST, 2011, p. 103) for stating event attributes, we have the following event attribut: $\#_{timestamp}$(e): the timestamp of events is given by the *effective semester* of an activity instance. The effective semester is computed as a difference from the academic term of the enrollment in the discipline and the first enrollment of the student. For example, if a student's first enrollment happens in the first semester of 2009 (academic term 2009-1), then all of the disciplines in which he enrolls in 2009-1 happen in the first effective semester. The disciplines in which that student enrolls in 2009-2 happen in the effective semester 2; and so on. With this classifier, we have no ambiguity in activity instances since there are no parallel instances of the same discipline for the same student.

We additionally have attributes that could comprise attributes like $\#_{resource}$ - we don't concern ourselves with resources in the current proposal, but we include the event attributes for reference. In the thesis, we describe how case attributes can be used to fundament the decision mining.

We consider students still enrolled in the program (even if not currently enrolled in any courses) as ongoing traces. In the extended representation we adopt for the model enhancement task the same is true for students that voluntarily adopt the *halted* status (students may, twice during the program, hold enrollments for up to four academic terms). We can't consider the effective termination of students that transfer to programs in other universities, as we do not have information on whether they graduate in a similar program or not. In our formulation, these are *halted* cases.

We now discuss the *challenges* posed by a domain like the academic program when viewed as a process, under the process mining definitions. These challenges motivate the usage of the academic program as a domain example.

In (VAN DER AALST, 2011, p. 112-114) several challenges are related to the extraction of event logs:

- Correlation: which events are of which case?

  We have no problem with the correlation between cases and events – the data relates the student enrollment (our case if) explicitly to each event.
- Timestamps – complete ordering of the events?

  We have explicit timestamps in the data. We compose plots (the traces) in ordered fashion from there.
- Snapshots – complete traces only?

  Our approach deals with partial traces naturally. One aspect of incomplete data that is indeed missing – and we ignore – is that students may *cancel* enrollment in disciplines. These cancellations don't figure in the data. This will impact, for example, in the rules for dropping out. The rules state that a student is dismissed from the program after failing *or cancelling the enrollment* in the same discipline five times. Another aspect that is not present in the data we considered is that if a student that *halts* the process returns to the program by re-enrolling, the *halted* status is changed to *enrolled.* Thus, the halting information for students that halt the program and return is not available.
- Scoping: required data reflected only?

In our approach, after anonymizing the students, we don't have additional case attributes other than the academic term of first enrollment. For the event attributes, we have attributes that we discard from the composition of the conceptual model – such as the *lecturer* of the disciplines' classes for that semester.

- Granularity: events of lower/higher levels?

  We assumed the level of disciplines as activities because that is what is reflected within our available data and corresponds to the expected analyses. However, it is apparent that a more fine-grained dataset could allow for the representation of intra-disciplines tasks. For example, at PUC-Rio, each discipline may comprise several evaluations (in the form of tests, projects or lab assignments, for example). The events could show the partial grades of each student in each of these evaluations. There are examples in the literature, particularly related to the usage of learning resources, of mining events of much finer, granularity – see Section 2.1.3.

Challenging characteristics of the domain

The educational domain used as an example in this thesis, when viewed as a process, presents the challenging characteristics that are tackled by the methods and algorithms implemented in the Library of Typical Plans for Process Mining.

- *intertask dependencies*

  The pre-requisites between disciplines are a clear example of intertask dependencies (supposedly and typically) enforced by the domain rules. The disciplines at PUC-Rio have *alternative* sets of pre-requisites, which relates to the next characteristic.

- *multiple dependencies*

  Not only do disciplines depend on sets of pre-requisites, there are *alternative* sets. We discuss the handling of alternative sets of preconditions throughout the thesis.

- *concurrent events*

  Another challenging aspect of the domain is that cases perform *multiple concurrent events*. It is typical for students to perform several disciplines at the same time.

- *failing activities*

  Students may not obtain passing grades in a discipline. The failure in disciplines has consequences in the trace, due to the intertask dependencies, the side-effects of failing activities in case-termination (e.g. students dropping out due to repeated failures) and that certain disciplines are required for graduation. This relates to the next characteristic.

- *repeated activities*
  In the educational domain students may not repeat disciplines at will – they are forbidden from re-enrolling in disciplines in which they have been approved, for example. However, they are implicitly obligated to re-enroll in disciplines in which they have failed if those are required disciplines for graduation.

- *knock-out structures*
  Finally, we deal with both partial-traces and terminated cases that may have completed the process in success (graduation) or failure (dropping out). We also consider students that have halted the process as a third kind of case-termination.

All of these characteristics are dealt with by our model. The representation of the domain in a conceptual model with a dynamic schema comprised of STRIPS-like operations eases most of the difficulties faced by graphical models in the representation of these characteristics.

One characteristic of the educational domain is that the academic terms are discrete and synchronous time periods. There is no intersection between academic semesters, for example – all disciplines happen during exactly one semester. The generalization of the method proposed to continuous time representations would imply several changes to the algorithms proposed, possibly requiring planning mechanisms apt to deal with more complex temporal logic constraints. The problem of planning in domains with time-oriented dynamics is well discussed in the literature (GHALLAB, NAU e TRAVERSO, 2004). Perhaps an approach such as Declare would be naturally more suited for domains with complex time relations.

A characteristic specific to the set of students we use for the experiments is that they are enrolled in an academic program with a high rate of dropped out students and failed disciplines. We've explored the characteristics of that program in comparison to other programs at PUC-Rio in (GOTTIN, JIMÉNEZ, *et al.*, 2017).

A final challenging characteristic of the domain is the changing of the domain rules over time. Particularly, the syllabus of the disciplines (that we use to determine the activity clauses), including their pre-requisites, are periodically updated. Not only the domain rules but also the recommended curriculum change periodically. We have circumvented part of the problems by considering in our examples sets of students with that enrolled around the same time, and defining the domain respecting the rules of that time period. However, not all information is retroactively available, and we were forced to use the current (as of the writing of this thesis) definitions of disciplines. As a result, the pre-requisite relations that we have considered might be different than the ones that were in place when the students were enrolled in the academic program. We have hinted at that issue when analyzing the discrepancies between the plots and the event log.