Sapienza University of Rome

Department of Computer, Control and Management Engineering
Master Degree of Engineering in Computer Science

A Planning-based Approach for Leveraging Activity Life Cycle Information in Declarative Trace Alignment

Thesis Advisor
Prof. Andrea Marrella

Research supervisor
Dr. Giacomo Acitelli

Candidate
Gianmarco Bordin
2081387

Academic Year 2023-2024

Abstract

In the world of Process Mining (PM), Trace Alignment is a fundamental practice that provides a way to measure how much the real world execution traces diverge from the expected behaviors defined in the model design phase. On the other hand, the Trace Alignment problem can be formulated as a planning task and AI-driven Automated Planning can be leveraged to explore the problem space, searching for optimal solutions. This can be beneficial for Business Process Management (BPM) that aims to improve the organization's processes. However, business process activities are still viewed as instantaneous events, which is a limitation, as they often span time and progress through various Life-Cycle states in real-world scenarios. This new approach employs Life-Cycle aware Trace Alignment exploiting AI automated planning power. The *Life-Cycle Aligner* tool can optimize this search by grounding the planning problems so that planners can find a solution efficiently. Our experiments show that this approach employs mid-to-high grounding time leading very small planner pre-processing and searching time in different problem settings.

Keywords: *Process mining, Trace Alignment, Automated planning, Business Process Management, Life-Cycle.*

Contents

Chapter 1

Introduction

From a functional perspective, an organization can be described in terms of how it transforms its inputs into outputs. Within this "black box", we can imagine a sequence of activities or procedures that contribute to this transformation, commonly referred to as a process.

These processes are composed of interconnected tasks or operations that work together achieving business objectives, such as delivering value to customers or meeting organizational objectives. To achieve it, organizations increasingly rely on Business Process Management (BPM). BPM enables businesses to optimize every stage of their product life-cycle from development and production to delivery and post-market support.

This approach not only reduces costs and increases quality but also ensures agility in meeting changing market demands. BPM is closely tied to Process Mining (PM), as the latter serves as a critical enabler for optimizing and enhancing business processes. PM is a discipline that bridges the gap between data analysis and business process management providing a foundation for informed decision-making within BPM. Each activity in this process can be recorded as an event, which represents a specific instance of an action or task being performed within the process at a certain time (described by its timestamp) and that can be recorded in an event log.

Conformance Checking in the context of PM refers to the analysis of how well the actual execution of processes aligns with a predefined process model. It helps organizations identify discrepancies between designed processes and their real-world execution.

Trace alignment, as the name suggests, is a technique used to align the possible deviations. It involves analyzing the sequence of events in the logs and matching them to the expected process flow to identify inconsistencies, such as missed steps or out-of-sequence activities. This process helps in identifying areas for process improvement.

There are different ways of process modeling, we can identify two main categories:

- Procedural Modeling, that states **how** the execution of a process should be carried out, specifying the sequence of activities that needs to be executed to achieve a specific goal.

- Declarative Modeling, where the model specifies **what** needs to be achieved without telling **how** it should be done.

In this thesis, we focus on Conformance Checking, and specifically on Trace Alignment with respect to Declarative Process Models. Declarative Process Models describe business processes by defining constraints that implicitly specify the permissible behaviors of the process. Declarative models are well-suited for dynamic environments where processes are highly flexible and subject to change. This is because, in such contexts, it is often difficult or impossible to specify all possible sequences of process executions. Instead, declarative models focus on defining the conditions that must hold true, reducing significantly the effort.
In this setting, any activity can be executed, as long as it adheres to the defined model constraints. We will focus on the well-known *DECLARE* process modeling language (van der Aalst et al., 2009) based on formal semantics derived from Linear Temporal Logic (Pnueli, 1977).

> According to De Giacomo et al. (2017), "We provide a sound and complete technique to synthesize the alignment instructions relying on finite automata theoretic manipulations. Such a technique can be effectively implemented by using planning technology. Notably, the resulting planning-based alignment system significantly outperforms all current state-of-the-art ad-hoc alignment systems. "

As stated in the article, we can correct the input trace of the event log by adding or removing events to resolve all the model discrepancies. The planner's output will provide a plan, consisting of a sequence of additions and deletions, that minimizes the cost of these corrections. Furthermore, as stated in Bernardi et al. (2016), in the current formulation, activities are still seen as atomic events and often in the real environments this is a limitation, because we have activities that pass through a series of life-cycle states. This thesis overcomes the limitations of the activity-atomic trace alignment perspective to provide a more realistic life-cycle-based representation. After outlining the aforementioned conceptual transition, we demonstrate how to encode the problem in a language that is compatible with the planner, so that it can find an optimal solution to the problem, namely the *life-cycle trace alignment problem*. We build upon this approach, taking it further by delving into an initial implementation that performs the encoding task. Thereafter, we develop an advanced tool that generates a grounded version of these planning problems, which allows us to significantly improve efficiency because the planner does not have to consider all parametric combinations of actions to perform in a given state but a restricted and essential set, allowing us to disruptively limit the state space, resulting in significantly shorter searching times. In this thesis we build upon the technique presented in (De Giacomo et al., 2017) to enable the completion of life-cycle-aware trace alignment.

1.1 Thesis Structure

The thesis is organized into six chapters, as follows:

- **Chapter 2**: Provides theoretical concepts and definitions needed to fully understand the thesis arguments. It starts by defining what is BPM and how it is crucial in the recent times. It continues the discussion describing what PM is and how it is related to Conformance Checking and Trace Alignment. The analysis proceeds, describing declarative models and highlighting *DECLARE* declarative process modeling language. The last topic treated is Automated Planning and how this research exploit it to reach the intended goal.
- **Chapter 3**: Presents traditional Trace Alignment (TA) and provides the transition to its life-cycle aware formulation.
- **Chapter 4**: Describes the tools developed to ensure a life-cycle correction based on a declarative model, starting from the general tool overview ending at the algorithm and implementation details.
- **Chapter 5**: Describes the experimental setup and provides the evaluation of the obtained results.
- **Chapter 6**: Provides a thorough review of the existing literature related to the study.
- **Chapter 7**: Concludes the thesis by summarizing the key findings, contributions of the research, and offering remarks on future directions.

Chapter 2

Background

In this chapter we will provide all the theoretical background needed to fully deepen into the concepts of this thesis. In particular, in section 2.1 we describe what Business Process Management actually is and why is it so important in today's organizations.
In section 2.2, we provide an overview of Process Mining and its role in Business Process Management, exploring key elements such as event logs, traces, and process models. Additionally, we define the significance of Trace Alignment in Conformance Checking for evaluating process behavior. In section 2.3 we introduce process modeling, emphasizing Declarative Process Modeling. Last but not least, in section 2.4 we emphasize the importance of Automated Planning as a tool for discovering solutions through an automation-driven approach.

2.1 Business Process Management

Business Process Management (BPM) is a methodology for managing and enhancing business processes. It focuses on the study of the business process life-cycle, which is a series of activities aimed at achieving a business goal.

2.1.1 Context and Definitions

BPM activities are enabled and supported by Information Systems, which provide the necessary tools and infrastructure for automating, managing, and refining business processes. It is essential to understand that Information is one of the most valuable resources for any organization. Information is different from Data. Data consists of raw facts, while information is a collection of data that has been organized and processed to provide additional value beyond the individual facts. Knowledge is the comprehension and insight gained from a collection of information to support a specific task or inform decision-making. The combination of knowledge and judgment enables the development of wisdom in addressing a specific situation. In this context, Information Systems play a crucial role in business operations by collecting, processing, storing, and disseminating data and information. Events, such as new orders or product deliveries, trigger business processes. These processes are governed by business rules that ensure compliance with organizational standards. Databases store data related to these events, enabling feedback mechanisms to monitor and improve the quality of information and ensure alignment with business objectives. Now we provide a more formal definition.

Definition 1 (Information Systems) *An information system (IS) is an integrated set of components designed to collect, store, process, and transmit data and digital information. It includes hardware, software, data, people, and processes that work together to convert raw data into valuable information, with the aim of supporting business goals.*

Business rules are closely related to Business Process Management (BPM) because they define and govern the behavior, decisions, and conditions within a business process.
To gain a more structured understanding of the terminologies and key concepts we provide some definitions.

Definition 2 (Business Process Management) *Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes (Weske, 2012).*

Definition 3 (Business Process) *Business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal, which is the objective that an organization aims to achieve by performing correctly its business processes (Weske, 2012).*

Moreover as outlined by Weske (2012), the business process life-cycle consists of interconnected phases, structured in a cyclical manner.

- **Process Identification**: In this phase, a business issue is identified, and the relevant processes are recognized and mapped. These processes are then connected, creating a process architecture that outlines their interrelationships.

- **Process Discovery** (also known as as-is process modeling): This step involves capturing the current state of each relevant process, typically by creating one or more as-is models that describe how processes are currently operating.

- **Process Analysis**: During this phase, issues within the as-is process are identified, documented, and, where possible, measured using performance indicators. The outcome is a structured list of problems, usually prioritized based on their impact and the effort needed to resolve them.

- **Process Redesign** (also called process improvement): The goal here is to propose changes that address the identified issues, helping the organization achieve its performance targets. The result is a to-be process model that serves as the blueprint for the next phase.

- **Process Implementation**: This phase focuses on executing the necessary changes to transition from the as-is to the to-be process. It encompasses two main components: organizational change management, which involves adjusting the way participants work, and process automation, which includes developing or upgrading IT systems to support the new process.

- **Process Monitoring and Controlling**: After the redesigned process is implemented, data is collected and analyzed to assess its performance against predefined objectives. Any bottlenecks, errors, or deviations from the desired behavior are identified, and corrective actions are taken. New issues may arise, prompting the cycle to restart as part of a continuous improvement process.

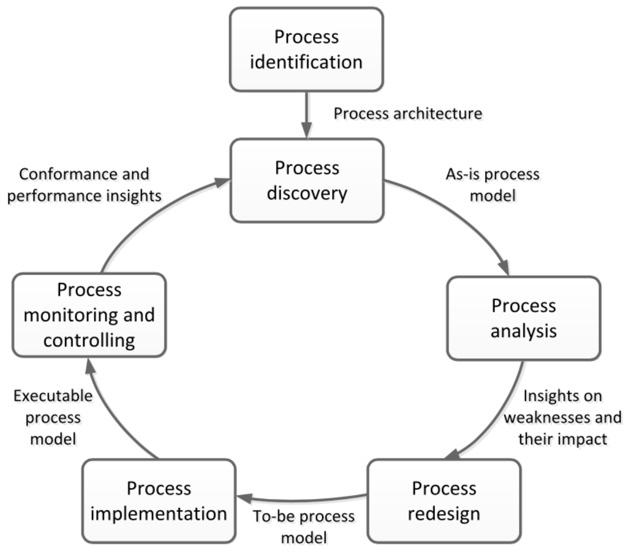


Figure 2.1: BPM lifecycle

Now that we have an overview of BPM and BP, we will refer to business processes simply as "processes" in the following discussion for simplicity. As stated in 3, we can describe a process as a composition of activities that jointly lead to a business goal. This is crucial for the subsequent discussion, as once we have fully grasped the concept of an organizational process, we will delve deeper into its components: events and activities, so that we can later introduce a central concept in this thesis: **the activity life-cycle**.

2.2 Process Mining

The concept of "Process Mining" was first introduced in 1998 within the field of software engineering by Cook and Wolf in their seminal work (Cook and Wolf, 1998). Since its inception, it has remained a significant focus of academic research and practical applications, continuing to be thoroughly studied and developed to the present day. Furthermore, interest in process mining has been rapidly increasing within the industry, driven by its potential to optimize operations and maximize profitability. Process Mining involves comparing event data (observed behavior) with process models (expected behavior) to identify discrepancies and gain insights. We now define Process Mining more formally mentioning the event log that will be later defined.

Definition 4 (Process Mining) *Process Mining is a technique used to analyze and improve business processes by extracting knowledge from event logs generated by information systems 1.*

The main goal of process mining is to gain insights into how processes are actually being executed in practice, as opposed to how they are assumed or designed to work. By analyzing the data stored in event logs, process mining enables organizations to ensure that processes are being carried out as intended. Next, we will review some key definitions necessary to understand the subsequent discussions.

Definition 5 (Activity) *An* activity *is a specific task or action that occurs within a process. It represents a unit of work executed as part of the process flow.*

$$a \in A$$

where A is the set of all possible activities in the process.

In process mining, activities are often denoted by their name and are part of the process model that describes the sequence of operations in a business process.

Definition 6 (Event) *An* event *represents a record of the execution of an activity within a given context, identified by its case id. It contains information such as the activity performed,* *a timestamp indicating when the activity occurred, and the associated case or instance to which the event belongs.*

$$e \in E \quad \textit{where} \quad e = (a, t, c)$$

where:

- *a is the activity executed,*
- *t is the timestamp of the event (indicating when the activity occurred),*
- *c is the case identifier (indicating the process instance to which the event belongs),*

and E is the set of all events.

Definition 7 (Trace) *A* trace *is a sequence of events that corresponds to a single process instance or case. It represents the ordered list of activities executed during the life-cycle of that process instance. The trace captures the specific path taken through the process model by that case.*

$$\tau \in T \quad \textit{where} \quad \tau = (e_1, e_2, \ldots, e_n)$$

where $e_i \in E$ is an event, and T is the set of all possible traces. The sequence τ corresponds to a single process instance, where the events are ordered according to their timestamps:

$$t(e_1) \leq t(e_2) \leq \cdots \leq t(e_n)$$

Now that we have all the necessary components, we can provide the definition of event log.

Definition 8 (Event Log) *An* event log *is a collection of recorded events that document the execution of activities within one or more process instances. Each event in the log corresponds to an individual occurrence of an activity, along with its associated timestamp and case identifier, as defined previously. An event log typically contains a set of traces, each representing the sequence of events for a particular process instance.*

$$L = \{\tau_1, \tau_2, \ldots, \tau_m\}$$

where L is the event log, and each trace $\tau_i \in T$ is an ordered sequence of events, as defined in 7. The event log thus captures the entire history of executed activities in one or more process instances.

A single execution of a process is recorded as a trace (or case). A trace consists of a sequence of events, where each event is associated with exactly one trace. The events within a trace are ordered by timestamps and may have additional attributes. Typical attributes include, but are not limited to, activity name, timestamp, cost, and resource.

Minimal requirement: The events within a trace must include at least the following attributes:

- Activity name.
- Case ID.
- Timestamp.

Table 2.1: A Sample event log fragment (Buijs, 2014).

Case ID	Event ID	Timestamp	Activity	Resource
10011	42933	2011/10/11 13:45	Confirmation of receipt	Resource21
	42935	2011/10/12 08:26	Check confirmation of receipt	Resource10
	42957	2011/11/24 15:36	Adjust confirmation of receipt	Resource21
	47958	2011/11/24 15:37	Check confirmation of receipt	Resource21
10017	43021	2011/10/18 13:46	Confirmation of receipt	Resource30
	43672	2011/10/18 13:47	Determine necessity of stop advice	Resource30
	43671	2011/10/18 13:47	Check confirmation of receipt	Resource30
	43674	2011/10/18 13:47	Adjust confirmation of receipt	Resource30
	43675	2011/10/18 13:47	Check confirmation of receipt	Resource30
	43673	2011/10/18 13:48	Determine necessity to stop indication	Resource30
	43676	2011/10/18 13:48	Adjust confirmation of receipt	Resource30
	43679	2011/10/18 13:51	Check confirmation of receipt	Resource30
	43686	2011/10/18 13:56	Adjust confirmation of receipt	admin2
10024	43229	2011/10/18 15:53	Confirmation of receipt	Resource03
	43727	2011/10/18 15:53	Check confirmation of receipt	Resource03
	43729	2011/10/18 15:53	Determine confirmation of receipt	Resource03
	43730	2011/10/18 15:54	Print and send confirmation of receipt	Resource03
	43728	2011/10/18 15:54	Determine necessity of stop advice	Resource03
	43731	2011/10/18 15:57	Determine necessity to stop indication	Resource03
10025	43231	2011/10/25 09:27	Confirmation of receipt	Resource03
	44375	2011/10/25 09:27	Check confirmation of receipt	Resource03
	44377	2011/10/25 09:27	Determine confirmation of receipt	Resource03
	44379	2011/10/25 09:28	Print and send confirmation of receipt	Resource03
	44376	2011/10/25 09:28	Determine necessity of stop advice	Resource03
	44380	2011/10/25 09:28	Determine necessity to stop indication	Resource03
10028	43250	2011/10/25 14:13	Confirmation of receipt	Resource03
	44467	2011/10/25 14:13	Check confirmation of receipt	Resource03
	44469	2011/10/25 14:13	Determine confirmation of receipt	Resource03
	44470	2011/10/25 14:14	Print and send confirmation of receipt	Resource03
	44468	2011/10/25 14:22	Determine necessity of stop advice	Resource03
	44471	2011/10/25 14:23	Determine necessity to stop indication	Resource03
	44472	2011/10/25 14:23	Report reasons to hold request	Resource03
	44473	2011/10/25 14:24	Check report Y to stop indication	Resource03
	44474	2011/10/25 14:24	Determine report Y to stop indication	Resource03
	44475	2011/10/25 14:25	Print report Y to stop indication	Resource03

Process modeling is closely related to the event log defined in 8 because, in process mining, event logs serve as the foundation for discovering and validating process models. Process modeling is the practice of representing business processes in a structured format, typically through diagrams or models. A process model is a visual or formal representation of the sequence of activities and decision points within a process. It serves as a blueprint for how a process is intended to operate. The main techniques used are:

- **Play-Out**: given a process model, it is possible to generate "behavior" multiple execution traces by simulating the process model.
- **Play-In**: that is the other way around, where multiple execution traces are taken as input to generate a process model.
- **Replay**: that takes both an event log and a process model as input, with the event log being replayed on the process model.

In process mining, the main tasks are generally divided into primary categories:

- **Process Discovery**, where Play-In is used to analyze multiple execution traces in order to construct a process model.
- **Conformance Checking**, where an event log is replayed (using Replay) on a process model to assess alignment and detect deviations.
- **Log Generation**, where a process model is used to generate synthetic event logs using Play-Out.
- **Other techniques** like Performance analysis, Process prediction, Process enhancement.

In the following section, we focus on Conformance Checking, as it is a key concept in this thesis and we provide a formal definition.

2.2.1 Conformance Checking

To better formalize what Conformance Checking is we need a couple of concepts to deal with. In process modeling we have a conceptual design of the model of a process and ideally we want that all the executions of this process, recorded in the log as a Trace, are **conformant** to that model. In principle, it is, but not in reality, where such a trace may contain deviations from the aforementioned model. We describe the amount of deviations as the amount of **noise** that the trace contains. Conversely, if we want to quantify how much a trace or an entire log is conformant to that model we use the term **fitness**. More formally speaking:

Definition 9 (Noise) *Noise refers to exceptional behaviors that do not accurately reflect the expected patterns of the process as modeled.*

Definition 10 (Fitness) *Fitness refers to the degree to which a model accurately reflects the behavior observed in the event log. A model exhibits perfect fitness if all traces in the log can be fully replayed from beginning to end without deviation.*

Now we can get exactly what Conformance Checking is: the process of comparing a discovered model with an event log to assess how well the model conforms to the observed behavior. This evaluation typically involves both global conformance measures and local diagnostics. Global Conformance Measures quantify the overall alignment between the model and the event log. For instance, a global conformance measure might reveal that 85% of the traces in the log can be successfully replayed by the model, indicating a high level of conformance across the entire log. In contrast, Local Diagnostics provide a more granular analysis by identifying specific points where the model and the log disagree. These diagnostics highlight mismatches at individual events or activities, such as when an activity (e.g., activity x) is executed more times than allowed by the model, pointing to local deviations that require further investigation. Together, these techniques enable a comprehensive assessment of the model's conformance to the actual process behavior, helping to identify both broad trends and specific discrepancies. Fitness measures refer to the proportion of behavior in the event log that is possible according to the process model. In the simple fitness computation, the replay of a trace is halted as soon as a problem is encountered, and the trace is marked as non-fitting. An event-based approach to calculating fitness, on the other hand, involves continuing the replay of the trace on the model.

2.2.2 Trace Alignment

Trace Alignment refers to the process of accurately matching the behavior observed in an event log with the process model, ensuring that each event in the log aligns with the correct activities in the model. This concept is closely tied to conformance checking, which assesses how well the model reflects the actual behavior. While conformance checking primarily focuses on identifying global and local discrepancies between the log and the model, trace alignment aims to pinpoint exact misalignments and computes an alignment that corrects all deviations. We provide some additional concept that will be used in the subsequent definitions. In the context of conformance checking and trace alignment, we can represent the model and the event log as two distinct automata. The model automaton represents the expected behavior of the system, while the log automaton reflects the observed behavior. These automata are compared to assess the alignment between the model's expectations and the actual process behavior. The model automaton consists of:

- **States in the model automaton**: Each state represents a specific configuration or condition of the system in the process. A state can denote the completion of a step in the process. A correct (with respect to the model behavior) process execution leads to accepting state(s).
- **Transitions in the model automaton**: A transition is a directed connection between two states, representing the flow of the process. Transitions are triggered by events or actions, defining the valid sequence of activities in the process.

The log automaton represents the actual behavior observed in the event log. The log automaton mirrors the structure of the model automaton but is based on the real-world executions captured in the log. It is composed by:

- **States in the log automaton**: These states correspond to the actual activities or steps recorded in the event log. They represent the real-world points reached during the process execution. A complete parsing of the log leads to accepting state(s).
- **Transitions in the log automaton**: These transitions represent the flow of the events or activities in the log, connecting states based on the real-world sequence of actions executed in the trace.

The goal of conformance checking is to compare these two automata (model and log automata) to determine how well the observed behavior (the event log) matches the expected behavior (the process model). The comparison involves finding the alignment between the two automata and evaluating the degree of conformance. A *move* refers to a transition between states in the process model or the event log. These moves help describing the evolution of traces through the model, and are classified into three types:

- **Move in the log**: A move in the log corresponds to an event or activity that occurs in the trace, reflecting an action or event recorded in the event log. It represents a step or action in the observed behavior of the process.
- **Move in the model**: A move in the model refers to a transition between states in the process model. It represents a valid action or event that the model allows at a specific point in the process. This move follows the rules and constraints defined by the model.

- **Synchronous move**: A synchronous move occurs when an event in the log automaton corresponds directly to a valid transition in the model automaton. This indicates that the behavior in the log is fully aligned with the expected behavior in the model.

In the process of trace alignment, the goal is to construct an alignment for a trace (or an entire log) by modifying the log, so that it conforms to the process model, moving between the log and model automata depending on the move. We start from the starting states of both automata and we want to reach the accepting state of both of them, such that the model is satisfied and the event log is completely parsed. In all the treatment, we consider zero cost for sync moves and unitary cost for each log or model move added. An alignment can either be optimal or worst-case, depending on the nature of the matches between the log and the model. A worst-case alignment:

- **(i)** All events in trace σ are converted to *log moves*, meaning that each event in the trace is mapped to a move in the log automaton.
- **(ii)** A shortest path from the initial state to the final state of the model is added, consisting of *model moves*. These model moves represent the transitions in the process model that are necessary to complete the process, but which do not correspond to any event in the trace.

The cost of the worst-case alignment is computed under the assumption that:

- There are *no synchronous moves*, meaning that no moves align between the log and model automata.
- Only *log moves* (from the event log) and *model moves* (from the process model) are considered, with no matching events between the log and model.

This alignment maximizes the deviation between the log and the model, resulting in the highest possible cost. In contrast, the optimal alignment minimizes the total cost of aligning the trace σ with the process model. In an optimal alignment:

- **Synchronous moves** are maximized, meaning that events in the trace σ are aligned directly with transitions in the process model. These synchronous moves represent the most aligned (and therefore least costly) way to match the log and the model.
- **Non-synchronous moves** (such as inserting log or model moves) are minimized in the optimal alignment, so as to reduce the overall cost of the alignment.

The cost of the optimal alignment is the minimum possible cost required to align the trace with the model, considering both synchronous and non-synchronous moves. An optimal alignment represents the best possible fit between the log and the model, with the fewest deviations. Now we can define the last concepts of this section regarding the computation of the fitness, in particularly fitness based on alignments.

$$\gamma_{\text{optimal}} = \begin{array}{|c|c|c|c|c|} a & b & c & d & e \\ \hline a & b & c & d & e \end{array} \quad \text{(Optimal Alignment)} \quad \text{and} \quad \gamma_{\text{misaligned}} = \begin{array}{|c|c|c|c|c|} a & b & c & \gg & e \\ \hline a & \gg & c & e & d \end{array} \quad \text{(Non-Optimal Alignment)}$$

Figure 2.2: Comparison of model and log alignment. The upper row represents the model, and the lower row represents the log. The "»" symbol is used when the log cannot be mimicked by the model (and vice-versa).

Definition 11 (Fitness based on alignments) *The value of the fitness of a trace can be calculated with the following formula:*

$$\mathit{fitness}(\sigma, N) = 1 - \frac{\delta(\lambda_N^{opt}(\sigma))}{\delta(\lambda_N^{worst}(\sigma))}$$

Where:

- σ*: The trace.*
- N*: The length of the trace.*
- $\delta(\lambda_N^{opt}(\sigma))$*: The cost of the optimal alignment of the trace* σ*.*
- $\delta(\lambda_N^{worst}(\sigma))$*: The cost of the worst-case alignment of the trace* σ*, where there are no synchronous moves and only moves in the model and log.*

The fitness score is calculated by comparing the optimal alignment to the worst-case alignment. The formula calculates fitness as the inverse of the ratio between the optimal alignment cost and the worst-case alignment cost. If the trace perfectly aligns with the model, the fitness will be 1 (perfect fitness). If the trace is as far from the model as possible, the fitness will be 0 (no fitness). More in general we can also define the fitness for an entire log.

Definition 12 (Fitness for the entire log) *The value of the fitness of an entire log can be calculated with the following formula:*

$$\mathit{fitness}(L, N) = 1 - \frac{\sum_{\sigma \in L} L(\sigma) \times \delta(\lambda_N^{opt}(\sigma))}{\sum_{\sigma \in L} L(\sigma) \times \delta(\lambda_N^{worst}(\sigma))}$$

Where:

- L*: The event log.*

- *N: The length of the trace.*
- *$L(\sigma)$: The number of occurrences of trace σ in the log.*
- *$\delta(\lambda_N^{opt}(\sigma))$: The cost of the optimal alignment of the trace σ.*
- *$\delta(\lambda_N^{worst}(\sigma))$: The cost of the worst-case alignment of the trace σ, where there are no synchronous moves, and only moves in the model and log are considered.*

This formula calculates the fitness of the entire event log by summing the costs of optimal alignments for all traces in the log, weighted by their frequency, and normalizing this sum by the total cost of the worst-case alignments for all traces. In the following section we provide a background on Declarative modeling and we will see more precisely how a process model can be designed.

2.3 Declarative Modeling

As previously mentioned, there are generally two primary categories of process modeling. Procedural Modeling needs to state all the possible sequences of activities to complete a process. It includes specifying the branch points and the exact flows in a specific process and more importantly, all the paths must be specified. Additionally, execution paths that are implicitly allowed but not explicitly modeled may inadvertently be excluded from the process. Moreover, sometimes processes are highly unstructured and it is not always possible to represent them with a Procedural model like *Petri-Nets* (Murata, 1989) or *BPMN* (Berlin, 2011). In certain cases, if the structure of the process is highly dynamic, it is not possible to apply changes to an existing model because it is too complex.
Similarly, overly restrictive constraints are often imposed, requiring designers to make assumptions that unnecessarily limit alternative execution paths. This results in processes becoming more rigid and less adaptable to variations that arise in practice. This first approach becomes cumbersome and hard to manage as the complexity of the process grows and it struggles to accommodate highly dynamic or unpredictable scenarios. Conversely, Declarative Modeling is more adaptive because it only needs to specify what needs to be satisfied regardlessly of the process size or complexity.
This approach divides possible process execution alternatives into the following categories:

- **Forbidden**: Executions that are explicitly disallowed and must never take place.
- **Optional**: Executions that, while generally undesirable, are still permissible.

- **Allowed**: Executions that align with expected and standard behavior.

We try to make a shift between the traditional paradigm in which "everything not explicitly modeled is forbidden" to a new approach where "everything not explicitly prohibited is allowed". The potential sequences of activities are implicitly defined by constraints, allowing any execution that does not violate these constraints. In the following part we will introduce *Linear Temporal Logic*. This logic that we use, allows us to formally specifying constraints of a declarative model.

2.3.1 Linear Temporal Logic over finite traces (LTLf)

In this paragraph we provide the concepts of Linear Temporal Logic over finite traces. LTL$_f$ (De Giacomo et al., 2013) are constructed from a set $\mathcal{P}$ of propositional symbols and include expressions formed using Boolean connectives: the unary temporal operator $\bigcirc$ (next-time), and the binary temporal operator $\mathcal{U}$ (until):

$$\varphi ::= A|\neg\varphi|(\varphi_1 \wedge \varphi_2)|(\bigcirc\varphi)|(\varphi_1\mathcal{U}\varphi_2) \tag{2.1}$$

with $A \in \mathcal{P}$.

Intuitively, $\bigcirc\varphi$ indicates that φ holds at the next instant, while $(\varphi_1\mathcal{U}\varphi_2)$ means that at some future point, φ_1 will hold, and until then, φ_2 will hold. Additionally, some common notation symbols are used:

- Boolean symbols, such as true, false, $\vee$ (or), and $\rightarrow$ (implies).
- *Last*, which represents $\neg\bigcirc$true, and refers to the last instant in the sequence.
- $\Diamond\varphi$, which is shorthand for true $\mathcal{U}\varphi$, and expresses that φ will eventually hold at some point before the last instant (inclusive).
- $\Box\varphi$, which is equivalent to $\neg\Diamond\neg\varphi$, and indicates that from the current instant up to the last instant, φ will always hold.

The semantics of LTL$_f$ is defined in terms of LT$_f$-interpretations, which are interpretations over finite traces representing a finite sequence of consecutive time instants. LT$_f$-interpretations are modeled as finite words t over the alphabet $2^{\mathcal{P}}$, meaning the alphabet consists of all possible propositional interpretations of the propositional symbols in $\mathcal{P}$. The following notation is used: The length of a trace π is denoted as $\mathit{length}(\pi)$. The positions, or instants, in the trace are denoted as $\pi(i)$, with $0 \leq i \leq \mathit{last}$, where $\mathit{last} = \mathit{length}(\pi) - 1$ is the final element of the trace. A segment (or sub-word) of π starting from position i and ending at position j is denoted by $\pi(i, j)$, where $0 \leq i \leq j \leq last$.

Given an LT$_f$-interpretation π, we can now define in an inductive way, when an LT$_f$ formula φ is true at an instant i (for $0 \leq i \leq last$), denoted as $\pi, i \models \varphi$, as follows:

- $\pi, i \models A$, for $A \in \mathcal{P}$ iff $A \in \pi(i)$.
- $\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$.
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$.
- $\pi, i \models \bigcirc\varphi$ iff $i < last$ and $\pi, i+1 \models \varphi$.
- $\pi, i \models \varphi_1 \mathcal{U} \varphi_2$ iff for some j such that $i \leq j \leq last$, we have that $\pi, j \models \varphi_2$ and for all k, $i \leq k < j$, we have that $\pi, k \models \varphi_1$.

We now provide some brief definitions:

- We say that a formula φ is true in π, in notation $\pi \models \varphi$, if $\pi, 0 \models \varphi$.
- We say that a formula φ is *satisfiable* if exists some LT$_f$-interpretation, in which φ is true.
- We say that a formula φ is *valid*, if it is true for all LT$_f$-interpretation.
- A formula φ logically implies formula φ', in notation $\varphi \models \varphi'$, if for all LT$_f$-interpretation π we verify that $\pi \models \varphi$ implies $\pi \models \varphi'$.

Each LTL$_f$ formula ϕ can be represented with a nondeterministic finite-state automaton (NFA) $\mathcal{A}$, which accepts exactly the traces that satisfy ϕ (De Giacomo et al., 2015). Formally speaking such automaton is denoted as a tuple $\mathcal{A} = \langle \sum, Q, q0, \rho, F \rangle$, where:

1. $\sum$ is the input alphabet;
2. Q is the finite set of automaton states;
3. q0 $\in$ Q is the initial state;
4. $\rho \subseteq$ Q $\times$ $\mathcal{P}$ $\times$ Q is the transition relation; and
5. F $\subseteq$ Q is the set of final states.

This representation translation involves several steps: beginning with an intermediate automaton (AFA), converting it into a nondeterministic finite-state automaton (NFA), and

then applying determinization techniques to convert the NFA into a deterministic finite-state automaton (DFA) and it involves in practice exponential time.
So far, we have introduced the basic concepts of LTLf and discussed how this logic can be translated into an automaton. This topic will be explored further in the following sections. This formalism provides an alternative way to describe the processes we will address next, including Declare templates.

2.3.2 Deterministic Finite Automaton (DFA)

In this section we provide some basic definitions to better understand what an automaton is. This concepts will be used in the subsequent part where we illustrates *Declare Templates.* The term is derived the Greek "autóuata" in english "self-acting". Firstly we start by defining what is an automaton.

Definition 13 (Automaton) *An automaton is an abstract, self-operating computational device that automatically follows a predefined sequence of operations.*

Intuitively an automaton is a theoretical computational model that performs tasks or processes automatically, based on a set of predefined rules or operations. It operates without external intervention, following a specific sequence of actions or steps that are determined in advance. We now define the Finite State Automaton that concretely is an automaton with a finite number of states.

Definition 14 *An automaton with a finite number of states is called a Finite Automaton (FA). It can be represented by a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where:*

- Q *is a finite set of states.*
- Σ *is a finite set of symbols, called the alphabet of the automaton.*
- δ *is the transition function.*
- q_0 *is the initial state from which any input is processed* $(q_0 \in Q)$*.*
- F *is a set of final states, which is a subset of* Q $(F \subset Q)$*.*

A Finite Automaton (FA) can be classified as either deterministic or non-deterministic:

- **Deterministic Finite Automaton (DFA)**: In a deterministic automaton, for each state and input symbol, there is exactly one transition to a next state. That is, for each state $q \in Q$ and input symbol $a \in \Sigma$, there is exactly one state $q' \in Q$ such that $\delta(q, a) = q'$.

- **Non-deterministic Finite Automaton (NFA)**: In a non-deterministic automaton, for a given state and input symbol, there may be multiple possible transitions, or no transition at all. That is, for each state $q \in Q$ and input symbol $a \in \Sigma$, the transition function δ can map to a set of states, i.e., $\delta(q, a) \subseteq Q$.

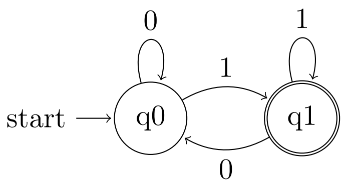


Figure 2.3: A simple DFA that accepts binary strings ending with "1".

In the following, we denote F as the set of accepting states, which refers to the subset of states in the automaton where the process or computation is considered successfully completed. If the automaton reaches any of these states after processing an input string, the input is accepted by the automaton. This means that, for an automaton to accept an input string, the string must lead the automaton to an accepting state from the set F after processing all the symbols of the input. Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, δ is the transition function, q_0 is the initial state, and F is the set of final states, we verify if an input string $w = a_1 a_2 \cdots a_n$ is accepted by the automaton as follows:

1. Start at the initial state q_0.
2. For each symbol a_i in the string w, update the state as:
   - **For DFA:** $q_{i+1} = \delta(q_i, a_i)$, where $q_i \in Q$ and q_0 is the initial state.
   - **For NFA:** $Q_{i+1} = \delta(Q_i, a_i)$, where $Q_i \subseteq Q$ and $Q_0 = \{q_0\}$.
3. After processing all symbols, check if the final state(s) are in the set of accepting states F:
   - **For DFA:** The sequence is accepted if $q_n \in F$.
   - **For NFA:** The sequence is accepted if $\exists q_n \in Q_{final}$ such that $q_n \in F$.

In the following part, we focus on DFA, noting that NFA can be transformed into an equivalent DFA. In particular we consider the symbols q_i of a string w as the activities that we find in a specific trace.

2.3.3 DECLARE

Now, we can integrate all the concepts: Declarative Modeling, LTLf, and DFA, to explain the Declare language and provide a comprehensive understanding of what a *DECLARE* template is.
DECLARE is a declarative process modeling language firstly introduced by Pesic and Van der Aalst in Pesic et al. (2007) and Pesic and van der Aalst (2006). Unlike imperative languages, which explicitly define the flow of execution among process events, a declarative language like *DECLARE* establishes a set of constraints that the process must adhere to throughout its execution.
DECLARE is designed to meet two key requirements:

- It must have a well-defined formal semantics.
- It should be representable in a way that is intuitive and understandable for end users.

A *DECLARE* model is defined by a set of constraints applied to activities, based on templates. Each template consists of three elements: a unique name, a LTLf formula that describes its semantics, and a visual representation (through DFA in our case). This part has the aim to provide several examples of *DECLARE* templates, including template names, the corresponding LTL_f and DFA, we concentrate on the templates that we will use in the following sections.

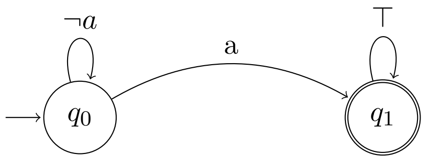
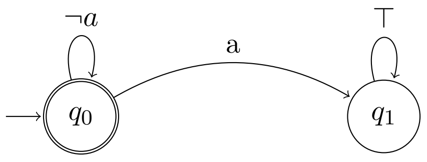
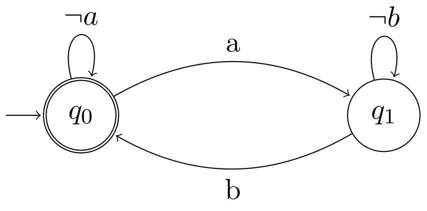
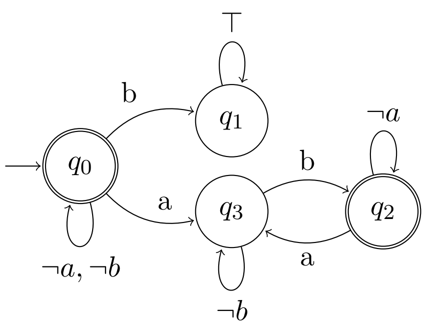
- The simplest templates are *existence* and *absence*. One is the opposite of the other in such a way that *existence* take as parameter one activity A and it declares that it occurs at least once and conversely *absence* requires that A never occurs in every process instance. *Existence* is considered activated (and satisfied) when the first occurrence of A appears in the trace. It is violated if no occurrence of A is found within the trace. For example, in the trace $\langle B, C, A, A, C \rangle$, the existence of A is satisfied because it appears in the trace. *Absence* is activated when an occurrence of A is found in the trace. Once activated, it is permanently violated. If no occurrence of A is found, the absence constraint is satisfied. For instance, in the trace $\langle B, C, C \rangle$, absence is satisfied because A does not appear.

- The *response* template specifies that when A occurs, B must eventually occur after A. It is activated whenever an occurrence of A is found in the trace. Once activated, the constraint is considered violated until an occurrence of B is found after A. If A does not occur, the constraint is always satisfied. For example, in the trace $\langle C, A, A, C, B \rangle$, A occurs in the process instance, and then B occurs after A, satisfying the constraint.

- The *succession* template specifies that activity A occurs if and only if it is followed by B (or vice versa) in the process instance. It is activated whenever an occurrence of A or B is found in the trace. Once activated, it is considered violated if there is no occurrence of A (or B) that occurs after (or before) B (or A). If not activated, the constraint is always satisfied. For example, in the trace $\langle C, A, A, C, B \rangle$, A is followed by B, satisfying the succession constraint.

- The *alternate succession* template specifies that A and B must alternate in the process instance, with each one following the other in the trace. It is activated whenever an occurrence of A (or B) is found. Once activated, it is considered violated until B (or A) is found after A (or B) and before another occurrence of A (or B) appears. If not activated, the constraint is always satisfied. For example, in the trace $\langle C, A, C, B \rangle$, A and B alternate in the process instance, satisfying the alternate succession constraint.

Table 2.2: Example of DECLARE templates.

Declare name	LTL$_f$ formalization	DFA
Existence(A)	$\neg\Diamond$A	
Absence(A)	$\Diamond$A	
Response(A,B)	$\square$(A $\rightarrow \Diamond$B)	
Succession(A,B)	$\square(A \rightarrow \Diamond B) \wedge$ $(\neg B \mathcal{W} A)$	 $\top$ b q_1 $\neg a$ q_0 b a q_3 q_2 a $\neg a, \neg b$ $\neg b$

- The *chain succession* template specifies that A and B must occur in the process instance only if B immediately follows A. It is activated whenever an occurrence of A (or B) is found in the trace. Once activated, it is considered violated if there is no occurrence of B (or A) immediately following A (or B). If not activated, the constraint is always satisfied. For example, in the trace $\langle C, A, B, A, B, C \rangle$, B occurs immediately after A, satisfying the chain succession constraint.

DECLARE also includes negative constraints to explicitly forbid certain activities from occurring. The *not responded existence* template states that if A occurs in a process instance, B cannot occur in the same instance. The *not response* template specifies that any occurrence of A cannot be eventually followed by B. The *not precedence* template indicates that any occurrence of B cannot be preceded by A. The *not succession* template indicates that A can never occur before B in the process instance. Finally, the *not chain succession* template specifies that A and B cannot occur in the process instance if B does not immediately follow A.

2.4 Automated Planning

Automated planning is an area of artificial intelligence (AI) that focuses on generating sequences of actions to achieve specific goals, given a set of constraints and an initial state. Planning systems autonomously devise strategies to navigate complex environments. This capability is vital in applications such as robotics, logistics, gaming, and autonomous systems.

2.4.1 Scenario

The challenge of building autonomous systems has been central to AI research since its inception, with the *control problem*, or *action selection problem*, being at its core. This problem involves determining the next action an autonomous system should take to achieve its goals in a given environment. Traditional solutions have often relied on **hard-coded controllers** that prescribe action sequences using high-level programming. While these approaches avoid the combinatorial explosion of possibilities during execution, they place a heavy burden on the programmer, requiring exhaustive specification of all possible scenarios. This rigidity can lead to biased solutions that overly constrain the search for actions, limiting adaptability and scalability in dynamic or complex environments. The modern AI approach reformulates action selection as a problem of **autonomous decision-making under uncertainty**. Rather than relying on static rules, AI systems leverage techniques such as:

1. **Automated Planning (AP)**: Systems reason about the future by generating action sequences that maximize goal achievement while adhering to constraints.
2. **Reinforcement Learning (RL)**: Agents learn optimal policies by interacting with environments, balancing exploration and exploitation.
3. **Hybrid Methods**: Combine planning and learning to adaptively select actions based on both long-term reasoning and real-time feedback.

There are two primary approaches in AI to address the challenge of autonomous behavior: First, the controller can be **learned from experience** (RL). This approach relies on the discovery and interpretation of meaningful patterns for a given task. However, learned solutions are often *black-box* in nature, making them difficult to interpret and verify.
Second, the controller can be **derived automatically from a model of the domain** (AP). This model incorporates the actions, current state, and goal, and is designed to be general and applicable across a wide range of tasks. However, a central challenge in planning is that *The Planning problem is computationally intractable, even for the simplest models, due to the exponential growth of the solution space.*
This is due to the fact that the action selection is computationally *hard* (NP-hard), primarily because the number of possible states grows exponentially with the size of the problem, which is determined by the number of problem variables. Specifically, if V is the set of problem variables and the search states are represented as *subsets* of elements from V, the total number of states is $2^{|V|}$.

One way to view the planning problem is as a path-finding task in a directed graph, where nodes represent search states and edges $s \to s'$ exist if an action a leads from state s to s'. Famous algorithms like Dijkstra's shortest path algorithm Dijkstra (1959) can find an optimal plan, but they typically run in polynomial time relative to the number of nodes, but here we have an exponential number of nodes and for this reason these methods are impractical for planning tasks. Despite the inherent complexity that limits a planner's ability to be truly *general-purpose*, planning research has made significant progress and we can identify two main reasons:

- **Heuristic Search**: Modern planners use heuristics to efficiently guide the search process by estimating the proximity to the goal, allowing for prioritization of promising states and significantly reducing the search space.
- **Domain-Specific Knowledge**: Planners are often applied to specific domains where domain-specific knowledge and constraints are leveraged, enabling more efficient plan-

ning by focusing on relevant actions and states rather than searching the entire state space.

Despite this inherent complexity, classical planners have proven capable of efficiently solving real-world problems, even those involving hundreds of propositions.

2.4.2 Classical Planning Model

In AI, **automated planning** is viewed as a *model-based approach* for automatically generating action plans to achieve specified goals. The process involves the following key components:

- **Model**: Represents the structure of the environment, actions, initial state, and goal.
- **Language**: A formalism used to express the model in a compact and interpretable form.
- **Planner**: An algorithm or search strategy designed to navigate the model and generate a feasible plan of actions.

The planning process takes the structured representation of the problem, expressed in a formal language, and uses algorithmic strategies to derive a plan that satisfies the given goal. Several classes of planning models exist, which are defined by the properties of the problems they represent. Key factors include:

- **Observability of the Current State**: The system may have *full* or *partial* observability.
- **Uncertainty in the Initial State**: The initial state may be *fully known* or only *partially known.*
- **Uncertainty in Action Dynamics**: Actions may have *deterministic* or *non-deterministic* outcomes.
- **Representation of Uncertainty**: Uncertainty can be expressed using *sets of states* or *probability distributions.*
- **Type of Feedback**: The system may receive *full*, *partial*, or *no state feedback.*

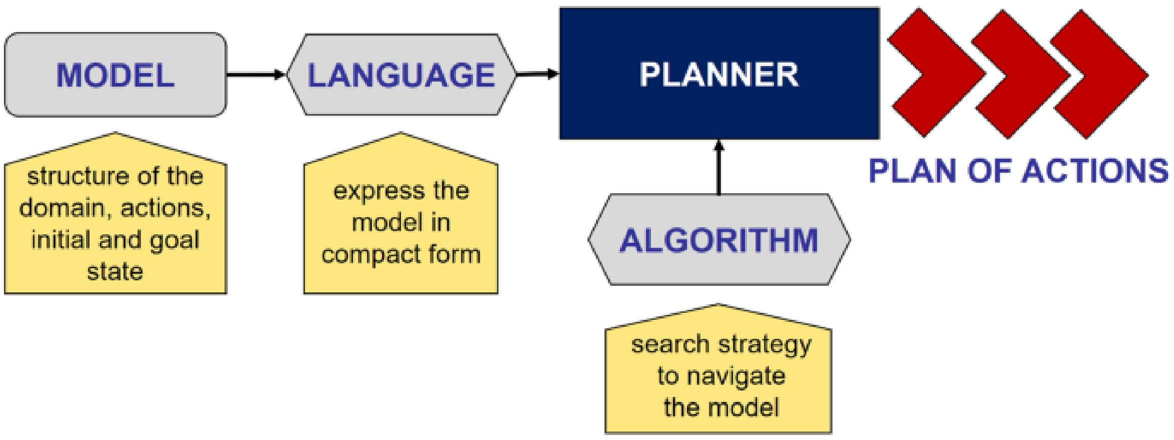

MODEL
LANGUAGE
PLANNER
PLAN OF ACTIONS
structure of the domain, actions, initial and goal state
express the model in compact form
ALGORITHM
search strategy to navigate the model

Figure 2.4: Automated Planning Model Overview.

In the following discussion, we focus on the classical planning model, where the system is assumed to have *full observability* of the current state, with the initial state being *fully known*. The outcomes of actions are considered *deterministic*, and uncertainty is not represented explicitly in the model. Additionally, the system receives *full feedback* about the state, enabling precise evaluation of its actions and progress towards the goal. It is composed by:

- A **finite and discrete state space** S.
- A **known initial state** $I \in S$.
- A **set of goal states** $S_G \subseteq S$.
- A set of **applicable actions** $A_s \subseteq A$ for each state $s \in S$.
- A **deterministic transition function** $s' = f(a, s)$, where $a \in A_s$.
- **Positive action costs** $c(a, s)$, where $c(a, s) > 0$.

Now we define more formally what is concretely a plan for a planning search problem.

Definition 15 (Solution or plan) *A **solution** (or **plan**) is a sequence of applicable actions:*

$$\pi = \langle a_0, a_1, \ldots, a_n \rangle$$

that transforms the initial state I into one of the goal states S_G.

The sequence of states $\langle s_0, s_1, \ldots, s_{n+1} \rangle$ satisfies:

- $s_0 = I$ *(the initial state),*
- $s_{i+1} = f(a_i, s_i)$ *for* $i = 0, \ldots, n,$

- *$a_i \in A_{s_i}$ for all applicable actions,*
- *$s_{n+1} \in S_G$ (the final state is a goal state).*

A plan is **optimal** if it minimizes the total cost of actions:

$$\sum_{i=0}^{n} c(a_i, s_i).$$

If all action costs $c(a, s) = 1$, the cost of the plan is equivalent to its **length**.

2.4.3 Heuristic Search

Heuristic search is a method used in AI planning to find solutions efficiently by guiding the search process with an evaluation function, known as a heuristic. A heuristic is typically a function that estimates the *cost* or *distance* from a given state to the goal state. It is used to prioritize which states to explore in order to find the solution faster than uninformed search algorithms. In heuristic search, the planning problem is treated as a search problem where the goal is to find a sequence of actions that transforms the initial state into the goal state. The key elements are:

- **State Space**: A graph where each node represents a state of the world, and edges represent the transitions between states as a result of applying actions.
- **Heuristic Function ($h(s)$)**: A function that estimates the cost of reaching the goal from a given state s. It is often domain-dependent and can vary in complexity and accuracy.
- **Search Algorithm**: A strategy used to explore the state space, such as A*, Greedy Search, or Best-First Search, where the heuristic function guides the selection of states to explore next.

Several algorithms make use of heuristics to find optimal or near-optimal solutions:

- **A* Algorithm**: Combines both the cost to reach a state (denoted $g(s)$) and the heuristic estimate to the goal $h(s)$. The evaluation function is:

  $$f(s) = g(s) + h(s)$$

  A* is complete and optimal if the heuristic is admissible (i.e., it never overestimates the true cost to the goal).

- **Greedy Best-First Search**: Uses only the heuristic function $h(s)$ to guide the search, without considering the cost to reach the state $g(s)$. It is faster but may not find the optimal solution.
- **Iterative Deepening A* (IDA*)**: A combination of the advantages of A* and depth-first search, where the algorithm performs depth-limited searches iteratively while maintaining an A* style heuristic.

Despite its power, heuristic search faces several challenges:

- **Heuristic Design**: Designing effective and admissible heuristics can be difficult, and poor heuristics may lead to inefficient search.
- **Computational Complexity**: The size of the state space can grow exponentially, and heuristic search may still suffer from computational intractability for large problems.
- **Overestimation of Cost**: If a heuristic is not admissible, it may lead to suboptimal solutions, or the algorithm may fail to find a solution at all.

2.4.4 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) is a formal language used to specify planning problems and domains in automated planning systems. It provides a standardized way to describe the objects, actions, initial states, and goals involved in a planning task. PDDL is crucial in enabling domain-independent planners to generate plans, as it separates the specification of the planning problem from the actual planner. This allows planners to be applied to various domains by simply altering the problem description, rather than redesigning the entire system, obviously always trading-off generality for efficiency. PDDL has become the de facto standard for expressing planning tasks and is widely used to benchmark different planning techniques and algorithms. A PDDL planning task consists of the following components: A PDDL planning task includes the following components.
Objects, which are the entities in the world that are of interest. **Predicates** describe properties or relationships of these objects that are relevant to the task, and they can have values of *true* or *false*. The **Initial State** provides a complete description of the world at the start of the planning process. The **Goal Specification** outlines the conditions or facts that we want to be true in the target state. Finally, **Actions/Operators** define the mechanisms for changing the state of the world. These actions are typically described in terms of **Preconditions**, which are the conditions that must be met for an action to be applicable, and **Effects**, which represent the changes that occur in the world as a result of the action being applied.

```
(define (domain logistics)
  (:types vehicle location package)
  (:predicates (at ?v - vehicle ?l - location)
               (loaded ?p - package ?v - vehicle)
               (delivered ?p - package ?l - location))

  (:action move
    :parameters (?v - vehicle ?from - location ?to - location)
    :precondition (at ?v ?from)
    :effect (and (not (at ?v ?from)) (at ?v ?to)))

  (:action load
    :parameters (?p - package ?v - vehicle ?l - location)
    :precondition (and (at ?v ?l) (at ?p ?l))
    :effect (and (not (at ?p ?l)) (loaded ?p ?v)))

  (:action unload
    :parameters (?p - package ?v - vehicle ?l - location)
    :precondition (and (at ?v ?l) (loaded ?p ?v))
    :effect (and (not (loaded ?p ?v)) (at ?p ?l)))
)
```

Figure 2.5: PDDL representation of the logistics domain, including actions for moving, loading, and unloading.

PDDL serves as a versatile and expressive language, enabling planners to operate by separating the domain from the problem. In other words, the domain is reused across multiple problems with different initial states and goals, while the problem can be easily defined for any specific scenario within that domain. Planning Problems in PDDL are expressed in two distinct parts:

- **PDDL Planning Domain (PD)**: Defines the available actions and predicates, providing an explicit representation of the world's structure and dynamics.
- **PDDL Planning Problem (PR)**: Specifies the objects, the initial state I, and the goal condition G.

```
(define (problem logistics-problem)
  (:domain logistics)
  (:objects truck1 - vehicle cityA cityB - location package1 - package)
  (:init (at truck1 cityA)
         (at package1 cityA))
  (:goal (and (at package1 cityB)))
)
```

Figure 2.6: PDDL representation of a logistics problem, specifying the initial state, goal state, and objects involved.

A planner that processes a problem encoded in PDDL is described as **domain-independent**. This means the planner can automatically generate a plan without requiring knowledge of the semantic meaning of the actions or domain. Its ability to function across diverse domains makes it a powerful and flexible tool for solving planning problems. This standardization has made PDDL a cornerstone for research and development in automated planning.

Chapter 3

Life-Cycle Trace Alignment

In 2, we explored the significance of Trace Alignment within the framework of Process Mining. In the opening segment, we describe the essential insights that enable the transition from the *traditional* trace alignment approach to the *life-cycle-aware* trace alignment introduced in this study.

3.1 From Traditional To Life-Cycle Aware Trace Alignment

Trace alignment is a process mining technique that compares process models with event logs to identify deviations in process executions. Identifies traces, sequences of activities, to reveal inefficiencies, ensure compliance, and uncover variations, enabling data-driven process improvements. Declarative Modeling handles the design of the process model and allows us to clearly define the expected behaviors that a specific process should follow.

The realizations of these models are captured in the Event Log. These logs contain a finite number of Traces each with an identifier and a variable number of additional attributes. Each trace holds one instance of the execution of a process. The fundamental building block is Event, which includes an *activity* name as one of its key attributes. Additional attributes may include the timestamp, resource, event ID, and *life-cycle* transition. Later in this chapter, we will see how to exploit this event information. In particular, in 3.1.4 we present how we leverage the activity event information that we find in the log, namely the *life-cycle* transition.

3.1.1 Traditional Trace Alignment Problem

In the traditional setting, we denote a trace $t = a,\ b,\ c$, where $a,\ b,\ c$ is the sequence of activities present in that trace. The elements of log traces are known as *events*. We represent the (finite) set of events as $\mathcal{E}$. Consider a log trace $t = e_1 \cdots e_n$ over $\mathcal{E}$ and an $LTLf$ formula

φ such that $t \not\models \varphi$ denoting that the trace does not satisfies the formula. Our goal is to make t satisfying φ so that it is "repaired". In formal terms, modifying t to form a new trace $\hat{t}$ such that $\hat{t} \models \varphi$. We assume that traces can be repaired using the following operations:

- *skip* an event (i.e., leave the event unchanged),
- *delete* an event at a given position in the trace t,
- *add* a new event at a certain position in the trace t.

The astute reader will note we can achieve modification of an event e by deleting it and adding a new event e' in that position. To represent these operations, we extend the set of events $\mathcal{E}$ by introducing a fresh events del_e and add_e for each $e \in \mathcal{E}$. These are called *repair events*, and the resulting set is denoted as $\mathcal{E}^+$. Traces over $\mathcal{E}^+$ are referred to as *repair traces* and a single repaired trace t is denoted as t^+. In a repair trace t^+, $e \in \mathcal{E}$ indicates that event e is skipped, del_e indicates that e is deleted, and add_e indicates that e is added to that specific trace t. A *repair trace* t^+ defines a sequence of modifications that, when applied, transform the original trace t into a new trace $\hat{t}$. We define a repair trace t^+ as *applicable* to a log trace t if t^+ can be obtained from t through the following operations:

1. Inserting any sequence of add_* events, where $*$ represents any event in $\mathcal{E}$, in any position in t;
2. Replacing any event e in t with either the event itself (i.e skipping e) or del_e, in any position in t.

We formally define the trace *induced by* t^+ as the result of applying the operations of the repair trace. This induced trace, $\hat{t}$, over $\mathcal{E}$ is obtained from t^+ by the following steps:

1. Removing every occurrence of del_e;
2. Replacing each occurrence of add_e with the corresponding event e.

For example, the trace induced by the repair trace $t^+ = del\_a\ add\_b\ del\_c\ c\ a\ add\_c$ is the trace $\hat{t} = b\ c\ a\ c$.
When t^+ is applicable to a trace t and induces the trace $\hat{t}$, we say that t^+ *transforms* t into $\hat{t}$. We have to define the last part of this approach that is the cost of an alignment. The *cost of a repair trace* t^+ is determined by summing the costs of all elements in a trace t^+, where each element's cost is given by the *cost function* $\mathcal{C}$.
For all cases, we define the following cost function:

Definition 16 (Cost Function) *Let* $\mathcal{C} : \{add\_*, del\_*, *\} \to \{0, 1\}$ *be a function such that:*

$$\mathcal{C}(a) = \begin{cases} 1 & \textit{if } a \in \{add\_*, del\_*\}, \\ 0 & \textit{if } a = *. \end{cases}$$

Then, the cost of a repair trace t^+ is given by the sum of the costs of its individual elements:

$$\text{cost}(t^+) = \sum_{a \in t^+} \mathcal{C}(a),$$

where a represents each element in the trace t^+, and $\mathcal{C}(a)$ is the cost of the corresponding action as defined above. The total cost is simply the sum of the costs of the actions that make up the repair trace. We address the *trace alignment problem*, which involves *finding a repair trace* t^+ *with minimal cost that transforms a trace* t *into a trace* $\hat{t}$ *such that* $\hat{t} \models \varphi$, given a trace t and an LTLf formula φ. It is clear that if φ is satisfiable, a solution always exists, as repair traces allow the transformation of any trace into another. The most naive solution consists by deleting all the trace events using a suitable long sequence of *del* events, followed by adding new events to generate the desired trace. The main challenge, however, lies in achieving this transformation at minimal cost. In 2 we have shown that a declarative model composed by different formulas can be equivalently represented as automata. In the following, the automata-based formulation is presented. We denote $\mathcal{A}$=$\langle \Sigma, Q, q_0, \rho, F \rangle$ as the corresponding automaton of the formula φ. We call it the *constraint automaton of* φ. The other automaton that we use is the *trace automaton of t* . It is defined as $\mathcal{T}$=$\langle \Sigma_t, Q_t, q_0^t, \rho_t, F_t \rangle$:

- $\Sigma_t = \{e_1, \dots, e_n\}$;
- $Q_t = \{q_0^t, \dots, q_n^t\}$ $n + 1$ states set;
- q_0^t is the initial state of the automaton;
- $\rho_t = \bigcup_{i=0,\dots,n-1} \langle q_i^t, e_{i+1}, q_{i+1}^t \rangle$;
- $\mathcal{F}_t = \{q_n^t\}$.

The set of traces accepted by $\mathcal{T}$, referred to as the *language* of $\mathcal{T}$, is denoted by $\mathcal{L}_\mathcal{T}$. In other words, the automaton *represents* the trace t. A sample automaton for the trace $t = a\ b\ c$ can be visualized in the provided picture.

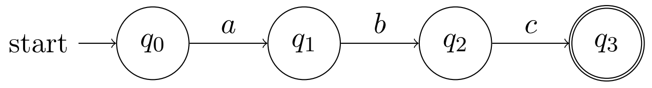


Figure 3.1: Example of a trace automaton.

Now we want to represent the augmented $\mathcal{T}^+ = \langle \Sigma_t^+, Q_t, q_0^t, \rho_t^+, F_t \rangle$ as we have done previously for t.

Definition 17 (Repair Automaton of a Trace) *Given an automaton $\mathcal{T}$, its augmented counterpart $\mathcal{T}^+ = \langle \Sigma_t^+, Q_t, q_0^t, \rho_t^+, F_t \rangle$ is an automaton where:*

- *ρ_t^+ contains all transitions from ρ_t, along with the following additional transitions:*
  - *A new transition $\langle q, del\_e, q' \rangle$ for every transition $\langle q, e, q' \rangle \in \rho_t$;*
  - *A new transition $\langle q, add\_e, q \rangle$ for every event $e \in \mathcal{E}$ and state $q \in Q_t$.*

We call $\mathcal{T}^+$ the repair automaton of t. *Notice that $\mathcal{T}^+$ accepts repair traces. The set of repair traces accepted by $\mathcal{T}^+$ is denoted by $\mathcal{L}_{\mathcal{T}^+}$.*

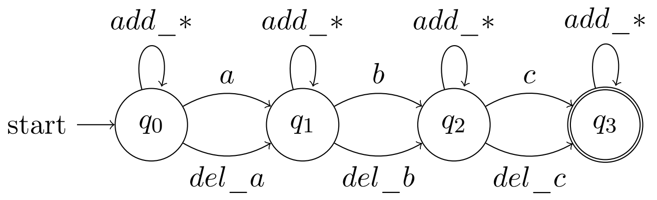


Figure 3.2: Augmented automaton of trace $t = a\ b\ c$ over $\mathcal{E} = \{a, b, c, d\}$.

The repair automaton T^+ of a log trace t over $\mathcal{E}$ is deterministic and accepts exactly all the repair traces t^+ over $\mathcal{E}^+$ that are applicable to t. Next, from an automaton $A = \langle \mathcal{E}, Q, q_0, \rho, F \rangle$, we derive the augmented constraint automaton of A, denoted by $A^+ = \langle \mathcal{E}^+, Q, q_0, \rho^+, F \rangle$, where ρ^+ contains the following transitions:

- A transition $\langle q, e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$;
- A transition $\langle q, add\_e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$;
- A transition $\langle q, del\_e, q \rangle$ for every $e \in \mathcal{E}$ and $q \in Q$.

Obviously Σ^+ must be similarly enhanced adding to the alphabet the del_e and add_e symbols. Intuitively, the automaton A^+ accepts all traces $\hat{t}$ that both satisfy the formula φ and are derived from the original trace t through repairs, where the modifications are explicitly indicated.

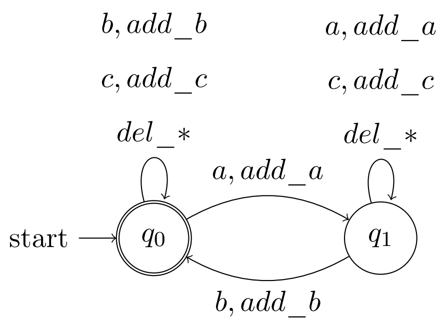


Figure 3.3: Augmented automaton of the formula $\varphi_1 = \Box(a \rightarrow \Diamond b)$ with alphabet $\{a, b, c\}$.

Notice that, from the previous definition, $L_A \subseteq L_{A^+}$, where L_A and L_{A^+} denote the languages of A and A^+, respectively. We can prove a stronger result: a trace t over $\mathcal{E}$ satisfies $t \in L_A$ if and only if $t \in L_{A^+}$. Furthermore, on every such trace, in both A and A^+ the same computation is performed. This follows from the fact that A and A^+ have the same initial, final, and non-final states, with the transitions of A being a subset of those in A^+. Intuitively, A^+ accepts all repair traces t^+ that induce a trace satisfying φ, as formalized in the following result.

Given an LTLf formula φ over $\mathcal{E}$, let A and A^+ be the corresponding constraint automaton and augmented constraint automaton, respectively. A repair trace t^+ over $\mathcal{E}^+$ is accepted by A^+ if and only if the trace $\hat{t}$ over $\mathcal{E}$, induced by t^+, is accepted by A.

By combining the previous results on the repair automata, we can conclude that, given a log trace t and an LTLf formula φ, the trace alignment problem reduces to searching for a repair trace t^+ (over $\mathcal{E}^+$) that is accepted by both A^+ and T^+ and has minimal cost. Acceptance by both T^+ and A^+ guarantees that we are considering only those repair traces t^+ applicable to t, and (ii) that induce a log trace $\hat{t}$ (that is to say, the found behavior) satisfying φ (the expected behavior). The main result is highlighted in the following theorem as discussed in De Giacomo et al. (2017).

Theorem 1 *Consider a log trace t (over $\mathcal{E}$) and an LTLf formula φ (over $\mathcal{E}$) such that $t \not\models \varphi$. Let T^+ and A^+ be the augmented automata constructed as explained above. If t^+, containing the minimal number of repair operations, is a trace accepted by A^+ and T^+, then it is a solution to the trace alignment problem of t.*

Clearly a trace $\hat{t}$ with minimal cost such that $\hat{t} \models \varphi$ can be obtained from t^+ by removing all occurrences of propositions of the form del e and replacing all occurrences of propositions of the form add e with e. Thus, given t and φ, the trace alignment problem boils down to identifying a trace that is accepted by both A^+ and T^+ ($t^+ \in \mathcal{L}_{\mathcal{T}^+} \cap \mathcal{L}_{\mathcal{A}^+}$) while minimizing the number of repair operations. In conclusion, we observe that Theorem 1 can be extended to accommodate multiple LTLf constraints $\varphi_1, \ldots, \varphi_n$. There are two ways of doing that as explained in De Giacomo et al. (2017). We can take the conjunction of these constraints and applying the previously outlined approach. Alternatively, as we will see later in this thesis, we can compute the augmented automata for each constraint, $A_1^+, \ldots, A_n^+$, and then search for a trace that satisfies T^+ and all of the automata $A_1^+, \ldots, A_n^+$. This forms the foundation for solving the trace alignment problem. Later we will see how to transform this formulation as a planning problem.

3.1.2 Life-Cycle model

In the previous section, we already provided the definition of the traditional problem which provides the basement for the subsequent arguments. Our goal is to overcome the limitations of the traditional setting where activities are treated as atomic/instantaneous events. This, because activities are normally composed of sub-states related between each others.
As we did for Declarative Modeling, we can equivalently visualize an *activity life-cycle* using automata. In particular, in the following picture we represent a first attempt to represent *life-cycle model* that we will assume during all the treatment of this thesis.

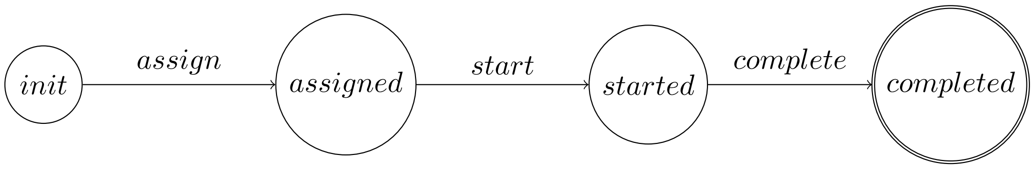


Figure 3.4: Example of a life-cycle automaton.

We represent our *activity life-cycle* using a dummy state that we will also call as *init state* or also *fake state*. Like the trace automaton and the constraint automaton 3.1.1 we have also a transition function. It has transitions only between adjacent states. For instance, the edge labeled as *assign* represents the fact that the activity has been assigned, so this would trigger a transition from the *init* state to the *assigned* state, stating that some resources may be instantiated. The arc labeled as *start* will trigger a similar transition but this time, stating that the life-cycle has entered the *started* state, meaning that some routine has started. Similarly to the previous edges the *complete* arc would trigger a transition to the *completed* state, effectively saying that this process/routine/procedure reaches its end and it is *correctly* executed. We note that, the final state, namely the *completed* state, is marked as accepting state. This means that we want our automaton to end in this state for a *correct* execution. We use the term *correct* to mean *the expected behavior with respect to our life-cycle model.*

This is not the final model that we will use in the following section but it is a first simple attempt to model a straightforward life-cycle and also it is simpler to understand. It has the main aim to provide the base insights to fully understand the following models.
We have represented the *correct* execution of a life-cycle process, but what if this life cycle does not end as we expect? This is frequently in *real-life* logs where the execution of a process does not align with a predefined model for this process. We already talked about Trace Alignment and we have seen how to cope with this. Now we want to represent this concept in another model, the *life-cycle* one. To face this issue we add another state to our first model, namely the *sink state*. We represent the changes in the provided picture.

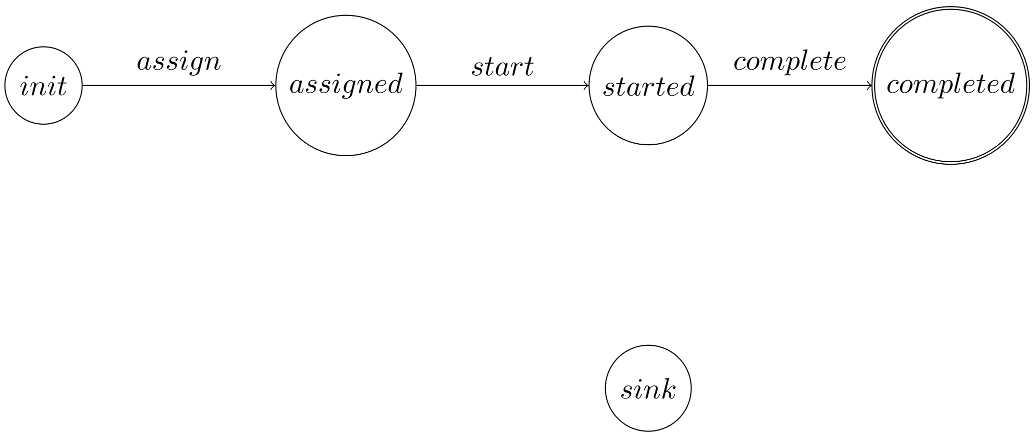


Figure 3.5: Example of a life-cycle automaton with a sink state.

Now the question would be how to reach this sink state from the already provided ones? We want our *life-cycle* model to be respected and any deviation to this will take from the *current* state to the *sink* state. The name of this newly introduced state is self-explanatory: *a state from where we cannot exit.* This means that we won't have outgoing edges from that sink state but only incoming arcs. The last question that we want to answer is what are the cases from which we enter this sink state? We remark that we want to follow the sequence $\langle assign, start, complete \rangle$. We define the set that contains the overall states in our life-cycle model as $\mathcal{Q}$. Whenever we are in a state $q \in \mathcal{Q}$ and we have an input that is different from the event that triggers to the next life-cycle state q', as we already saw in the first automaton, we go to the *sink* state. We make clear those insights provide a graphical representation in the following automaton.

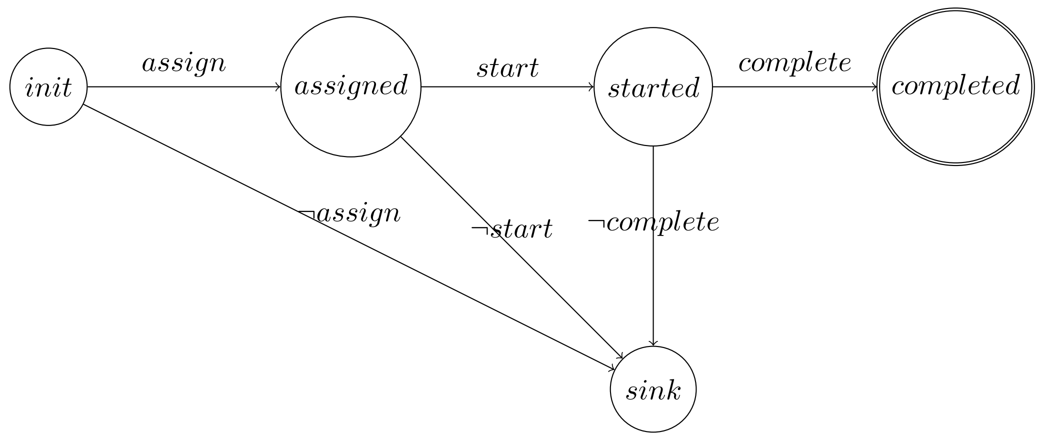


Figure 3.6: Example of a life-cycle automaton with a sink state and incoming sink edges.

The symbol ¬ represents the logical Boolean *not*. Other types of representations may focus solely on transitions triggered by specific events, while implicitly assuming that all other transitions are disallowed, effectively causing the automaton to remain in the same state. Alternative visualizations may include self-loop edges for those cases. Our formulation is actually explicit, meaning that we represent all the possible transitions excepts for the self-loop ones.

Up to now we have represented an automaton that models effectively the expected sequence *assign,start,complete* that we want to be respected in our formulation because whenever we are in a state and the input event is not the next in the sequence, for instance we are in the assigned-state and the input event is different from the start event, we effectively move to the sink state and we end the computation leaving the automaton in a non-accepting state, this is exactly what we wanted to accomplish. Conversely, if the event is the expected one we move to the next state. If we arrive to the completed state, we are done and the automaton ends in an accepting state. At this time we have correctly modeled a generic life-cycle that takes into account only non interleaving life-cycles. For example if we had this sequence *assign,start,assign,start,complete,complete* the automaton would end in a non accepting state. The automaton that we consider during all the formulation of this thesis will not consider the case for which multiple life-cycles, *of the same activity*, interleave, we will clarify this in the latter sections.

We consider another example to clearly understand the last modifications that we want to apply to our automaton: if we see the sequence *assign,start,complete,assign,start,complete* the automaton must end in an accepting state. To face this problem we make the following modifications.

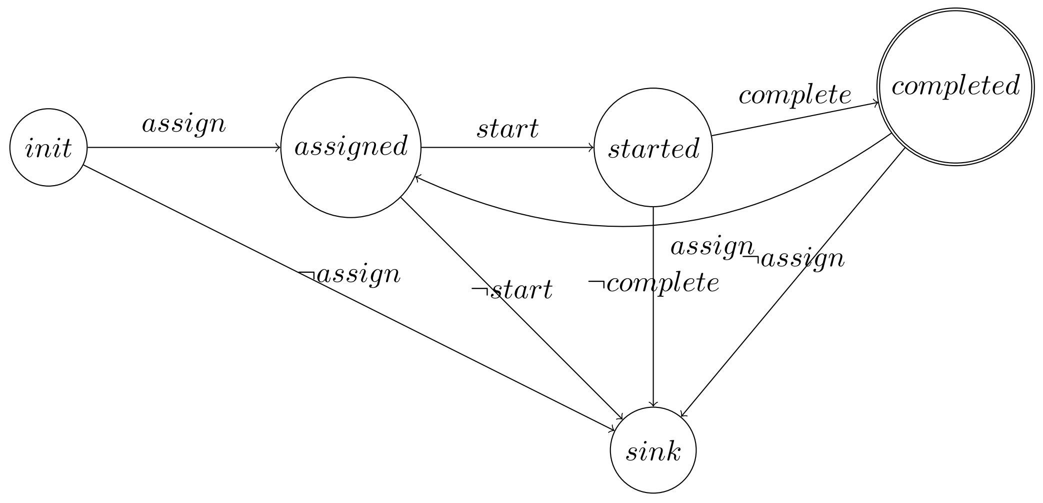


Figure 3.7: Example of a subsequent-life-cycles automaton.

After this change we correctly model the overall life-cycle process in our diagram. In this way, subsequent life-cycles sequences can be represented and tracked by our automaton, enhancing our representation power. We have previously mentioned that in our discussion we will not consider *interleaved* life-cycles of the same activity, this assumption will simplify the modeling of the life-cycle automaton. In the subsequent subsection we define better what an activity life-cycle is in our context and how to link it with the logs, events and activities.

3.1.3 Problem Definition

Up to now, our life-cycle model is clearly defined, but we want our modeling to be related to an activity: we want to model the *activity life-cycle*. This means, that the activity must be mentioned within this design. For simplicity, we consider the existence of an activity named a in the event log, this lead us to modify our automaton in the following.

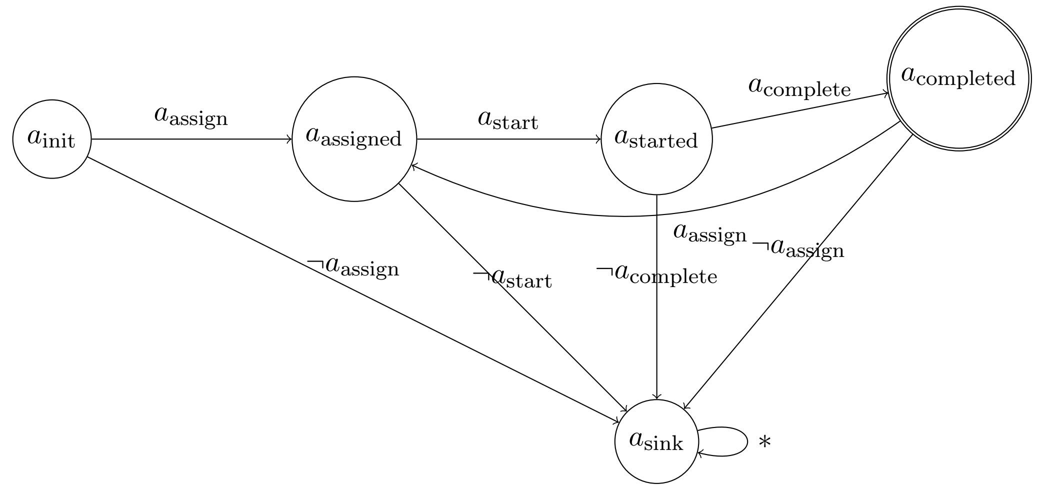


Figure 3.8: Example of an activity-life-cycle automaton.

Where $*$ symbol represents any term $\epsilon \in \mathcal{E}$. To simplify the notation, we will refer to the *subsequent-life-cycles automaton* simply as the *life-cycle automaton*, while keeping in mind the full range of scenarios it is capable of modeling. For the same reason we will often make the notation lighter, by referring to each state as simply its name, avoiding the activity name. Now that we have defined the activity *life-cycle* model, we have to state how to actually enforce this life-cycle.

In 3.1 we have seen how models can be used to effectively align the trace with respect to the expected behaviors: each constraint has its own automaton that has to be respected. Now we arrive to the following question, *where to employ this activity life-cycle?* We can consider this automaton has an *additional constraint that have to be satisfied.*

We will provide a clearer and more detailed explanation in the following subsection on how this can be effectively implemented. The last thing that we want to point out in this part is how this *activity life-cycle information* can be collected from the event logs. We have mentioned that in most event logs an event entry has several information within it. We collect the content of the *life-cycle transition* entry and we concatenate it with the *concept name* entry value constructing the so-called *life-cycle activity.* For example, if the content of the concept name entry is *a* and the content of the life-cycle transition entry is *complete* our *life-cycle activity* will be *a-complete.*

Furthermore, the concept of the traditional *general* activity *a* is overloaded, in the sense that, it will be composed by its life-cycles elements, namely *a-assign,a-start,a-complete.* The advantage of this is that we have more expressive power because we can put constraint on the traditional *general activity a* declaring it on the *a-complete life-cycle activity* and in the meanwhile, we can also declare *life-cycle constraints* putting it on all the life-cycle activities, namely *a-assign,a-start,a-complete.* We accomplished the goal, up to this time only theoretically, that we had described in the preamble and this will lead to the definition of the Life-Cycle Aware Trace Alignment defined in the following subsection.

3.1.4 Life-Cycle Aware Trace Alignment

Differently from the traditional setting, each general activity becomes a life-cycle activity. Each activity now has a life-cycle. Consequently, an activity a, from the traditional notation, will be now denoted with its subscript representing the life-cycle information. This means that a general activity a will be composed of several life-cycle events, namely a_{assign}, a_{start}, $a_{complete}$, effectively overcoming the limitations of the traditional activity-atomic approach.

An instance trace t could be $t = a_{assign},\ b_{assign},\ c_{complete}$, where $a_{assign},\ b_{assign},\ c_{complete}$ is the sequence of activities present in that trace. The log traces elements are the same as before, but now we will consider the life-cycle information. The activity name, in the corresponding event entry, will be considered together with its life-cycle tag. All events are contained in $\mathcal{E}$. In the trace alignment explanation in 3.1, we have provided the simplest base case with only one constraint formula φ. The case with a single formula φ can be easily extended to the more general one with more constraints. To avoid misunderstandings, by now, when we write φ we refer to the set $\varphi = \{\varphi_1 \cdots \varphi_c\}$, where each φ_i is a single LTLf formula. From now we will refer to model formulas that model non-life-cycle constraints simply with *model* formulas. Consider a log trace $t = e_1 \cdots e_n$ over $\mathcal{E}$, $LTLf$ formulas denoted as $\varphi = \{\varphi_1 \cdots \varphi_c\}$, where each $\varphi_i \in \varphi$ with i $= 1 \cdots c$ and c the total number of model constraints. Let the life-cycle formulas be denoted as $\vartheta = \{\vartheta_1 \cdots \vartheta_l\}$, where each $\vartheta_i \in \vartheta$ with i $= 1 \cdots l$ and l the total number of life-cycle constraints. Each ϑ_i is a formula over life-cycle activities with subscripts that can assume any value in $\{assign, start, complete\}$. Conversely, for the model formulas regarding general activities, that we now consider together with the *complete* subscript, the formula will be over life-cycle activities with subscript *complete*. The formulas may not be already satisfied, for this reason we need the trace alignment. To state that the formulas are not satisfied we write, as before, $t \not\models \varphi$ or $t \not\models \vartheta$. Another time our goal is to make t satisfying φ and ϑ so that it is "repaired" both for model constraints and life-cycle constraints. In formal terms, modifying t to form a new trace $\hat{t}$ such that $\hat{t} \models \varphi$ and $\hat{t} \models \vartheta$. As before, We assume that traces can be repaired using the following operations:

- *skip* an event (i.e., leave the event unchanged),
- *delete* an event at a given position in the trace t,
- *add* a new event at a certain position in the trace t.

As in the traditional settings, we extend the set of events $\mathcal{E}$ by introducing fresh events, but now they will include the life-cycle activities: $del\_a_{life-cycle}$ and $add\_a_{life-cycle}$ for each $a \in \mathcal{U}$. As before we denote the transformed trace as $\hat{t}$ that can be obtained from t using the following operations:

1. Inserting any sequence of *add_ ** activities, where $*$ represents any activity in $\mathcal{E}$, in any position in t;
2. Replacing any activity a in t with either the activity itself (i.e skipping a) or del_a, in any position in t.

The induced trace, $\hat{t}$, over $\mathcal{E}$ is obtained from t^+ by the following steps:

1. Removing every occurrence of del_a;

2. Replacing each occurrence of add_a with the corresponding activity a.

The cost of a repair trace t^+ is given by:

$$\text{cost}(t^+) = \sum_{a \in t^+} \mathcal{C}(a),$$

Let a represent each element in the trace t^+, and let $\mathcal{C}(a)$ denote the cost of the corresponding action as previously defined. The total cost of the repair trace is computed as the sum of the costs of all actions within the trace.

We study the *trace alignment problem*, which is defined as *finding a repair trace t^+ with minimal cost that transforms a given trace t into a trace $\hat{t}$ such that $\hat{t} \models \varphi$ and $\hat{t} \models \vartheta$*, where t is a trace, φ is the model formula set and ϑ is the life-cycle model formula set. Clearly, if φ and ϑ is satisfiable, a solution always exists. As usual we provide also the more intuitive automata-based formulation. In addition to the Traditional Trace Alignment Problem and Traditional Trace Alignment Problem defined in 3.1, we construct an automaton for each life-cycle formula that must be respected. We refer to a single life-cycle formula as ϑ_i and we avoid to put the ϑ_i subscript in the automaton formulation to avoid notation overloading. We denote $A_{life-cycle} = \langle \Sigma_{life-cycle}, Q_{life-cycle}, q_{0_{\text{life-cycle}}}, \rho_{life-cycle}, F_{life-cycle} \rangle$ as the automaton of the formula ϑ_i. We now provide a more formal definition.

Definition 18 (life-cycle enforcement) *Let $\mathcal{U}$ denote the set of all activities observed in a given event log. Formally, $\mathcal{U}$ represents the complete set of unique activity labels, recorded across all traces within the log, noted as L. For each life-cycle constraint ϑ_i we construct a life-cycle automaton noted as $\mathcal{A}_{\textit{life-cycle}}$.*

We call it the *life-cycle constraint automaton of ϑ_i*.

From the automaton $A_{life-cycle} = \langle \Sigma_{life-cycle}, Q_{life-cycle}, q_{0_{\text{life-cycle}}}, \rho_{life-cycle}, F_{life-cycle} \rangle$, we derive the augmented life-cycle constraint automaton of $A_{life-cycle}$, noted by $A^+_{life-cycle} = \langle \Sigma^+_{life-cycle}, Q_{life-cycle}, q_{0_{\text{life-cycle}}}, \rho^+_{life-cycle}, F_{life-cycle} \rangle$, where $\rho^+_{life-cycle}$ contains the following transitions:

- A transition $\langle q, e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$;

- A transition $\langle q, add\_e, q' \rangle$ for each transition $\langle q, \psi, q' \rangle \in \rho$ and $e \in \mathcal{E}$ such that $\{e\} \models \psi$;

- A transition $\langle q, del\_e, q \rangle$ for every $e \in \mathcal{E}$ and $q \in Q_{life-cycle}$.

We know that our aim is ensuring that, given a trace t, a specification φ_i, a specification ϑ_i the objective is to identify a trace t^+ such that:

$$t^+ \in \mathcal{L}_{\mathcal{T}^+} \cap \mathcal{L}_{\mathcal{A}^+ \cup \mathcal{A}^+_{\text{life-cycle}}},$$

where $\mathcal{L}_{\mathcal{T}^+}$ is the language of repaired traces for t, and $\mathcal{L}_{\mathcal{A}^+ \cup \mathcal{A}^+_{\text{life-cycle}}}$ is the language accepted by the automaton representing the union of the φ_i and ϑ_i formula constraints. This base case can easily extended to formula sets φ and ϑ. As usual, the solution minimizes the number of repair operations required to transform t into t^+. In the following section, we take a step closer to the final formulation by presenting the planning problem underlying our approach. This formulation is essential for leveraging automated planning to address the life-cycle-aware trace alignment problem.

3.2 Planning problem Formulation

To leverage automated planning as a tool for solving our problem, we must first formulate the life-cycle trace alignment as a planning problem. We model the events $e \in \mathcal{E}^+$ as actions. The execution of this actions (i.e, by the planner), triggers transitions between states in $\mathcal{T}^+$, $\mathcal{A}^+$ and $\mathcal{A}^+_{life-cycle}$ automata. We can execute actions according to the automata current states. An action that we note as a is executed only if there exist outgoing transitions, with label a, from the current states of $\mathcal{T}^+$, $\mathcal{A}^+$ and $\mathcal{A}^+_{life-cycle}$ automata. Our goal is to find a deterministic plan that, starting from the initial states of $\mathcal{T}^+$, $\mathcal{A}^+$ and $\mathcal{A}^+_{life-cycle}$ automata, leading all the automata to their final accepting states with a minimum cost. The only actions that we can perform are *add*, *del* or *sync* moves. Actions *add* and *del* have unitary cost while *sync* move has zero cost. At the end we obtain a deterministic plan for our planning problem that is a representation of the repair trace solving our quest. The resulting plan represents the repair trace that solves the problem, specifying how each event in the input trace must be modified to meet the constraints with the minimum cost. More formally we provide the required definitions.

Definition 19 (Planning domain) *A (deterministic) planning domain with action costs, over a set of propositions* Prop*, is defined as the tuple* $D = \langle S, A, C, \tau \rangle$*. The components of this tuple are:*

1. $S \subseteq 2^{\mathsf{Prop}}$ *is the finite set of states for the domain, viewed as propositional interpretations.*

2. *A is the finite set of actions for the domain.*

3. *$C : A \to \mathbb{N}^+$ is the cost function.*

4. *$\tau : S \times A \to S$ is the transition function.*

A plan for D can be defined as a finite sequence $\pi \in A^*$ of actions, namely $\pi = a_1 \cdots a_n$ with each $a_i \in A$ for i = 1$\cdots$ n. The plan is said to be executable from a state $s_0 \in S$ if there exists a states sequence $\sigma = s_0 \cdots s_n$ such that, for $i = 0, \ldots, n-1$, $s_{i+1} = \tau(s_i, a_{i+1})$. The cost of π on D is defined, as usually, as:

$$C(\pi) = \sum_{i=1}^{n} C(a_i).$$

Now that we have defined the problem domain and what a plan formally is, we start to define the *cost-optimal planning problem.*

Definition 20 (Cost-optimal planning problem) *A cost-optimal planning problem is a tuple $P = \langle D, s_0, G \rangle$, where:*

- *D is the planning domain.*

- *$s_0 \in S$ is the initial state of the problem.*

- *G is the problem goal.*

Our plan π will be a solution for the planning problem, noted as P, if for last state: $s_n \models G$. A solution π to P is optimal if, for all the solutions π', such that $\pi' \neq \pi'$, we have $C(\pi) \leq C(\pi')$. With the formal definition of the planning problem established, we can now explore how trace alignment—specifically life-cycle-based trace alignment—can be effectively reduced to cost-optimal planning.

For an instance of life-cycle trace alignment with a trace t, a model formula φ_i and a life-cycle formula ϑ_i, the corresponding planning problem can be derived doing the following:

1. We compute the automata $T^+ = \langle \Sigma_t^+, Q_t, q_{t0}, \rho_t^+, F_t \rangle$, $A^+ = \langle \Sigma^+, Q, q_0, \rho^+, F \rangle$ and $A^+_{life-cycle} = \langle \Sigma^+_{life-cycle}, Q_{life-cycle}, q0_{\text{life-cycle}}, \rho^+_{life-cycle}, F_{life-cycle} \rangle$.

2. We define the planning domain $D = \langle S, A, C, \tau \rangle$, where: $S \subseteq 2^{Q_t \cup Q \cup Q_{life-cycle}}$ (we recall that automata states are treated as propositions and that repair propositions are treated as actions).

3. We define the cost function as follows. For all $e \in \Sigma \cup \Sigma_{\text{life-cycle}} \cup \Sigma_t$, we have:

$$C(\text{sync } e) = 0, \quad C(\text{del } e) = 1, \quad C(\text{add } e) = 1.$$

4. We define the transition function τ as follows. For all $a \in A$, $q_{\text{t}}, q'_{\text{t}} \in Q_{\text{t}}$, and $Q_1, Q_2 \subseteq Q \cup Q_{\text{life-cycle}}$, we have:

$$\tau(\{q_{\text{t}}\} \cup Q_1, a) = \{q'_{\text{t}}\} \cup Q_2 \quad \text{if and only if all of the following conditions hold:}$$

   (a) $(q_{\text{t}}, a, q'_{\text{t}}) \in \rho^+_{\text{t}}$;

   (b) There exists a transition $(q, a, q') \in \rho^+$ such that $q \in Q_1$;

   (c) For all $q \in Q_1$, if there exists a transition (q, a, q'), then $q' \in Q_2$;

   (d) For all $q' \in Q_2$, there exists a transition (q, a, q') such that $q \in Q_1$.

This formulation aims to represent the synchronous execution of T^+, $A^+_{life-cycle}$ and A^+ over an input trace t. The actions discussed before trigger, under certain conditions, transitions to successor states for these automata. We have already mentioned that this *repairs*, to the original trace t, are mapped to synchronization actions indicating no change, add actions representing additions, and del actions corresponding to deletions, all of these actions have a defined cost as expressed above.
Obviously, with the aim of finding an optimal plan, *sync* action are the preferred ones. The execution of those actions triggers automata progresses and we want to reach their final accepting states, starting from their initial states, so that the trace is repaired. Finding a minimal cost plan results in an optimal solution to the life-cycle trace alignment problem. Based on these observations, we define the cost-optimal planning problem $P = \langle D, s_0, G \rangle$ as follows:

$$s_0 = \{q_0, q^t_0\}, \quad G = q^t_f \wedge \bigwedge_{q \in F \cup F_{life-cycle}} q$$

where $q^t_f \in F_t$ (note that T^+ has a unique final state).

3.3 PDDL Encoding

Having a defined planning problem formulation allow us to make a step further to the implementation of the actual solution. In this part, we focus on encoding our life-cycle trace alignment planning problem in *PDDL*. The **Planning Domain Definition Language (PDDL)** is a standardized language used in the field of Artificial Intelligence (AI) for specifying planning problems and domains. It was introduced in 1998 during the International Planning Competition (IPC) (McDermott et al., 1998). PDDL allows for the definition of:

- **Domains:** These describe the general rules, object types, and actions available in a planning environment.
- **Problems:** These specify specific instances of planning scenarios, including the initial state of the world and the desired goal state to be achieved.

One of the main advantages of this formulation is that it allows the separation of the planning domain and problems. Thanks to this, different problems instances can be used with the same domain, avoiding to rewrite the planning domain. This modularity decouples the two modules allowing for greater flexibility. Despite this, in the following chapters, we will observe that this decoupling comes at a cost because planners must often consider a larger search space and handle more generic constraints, which can lead to longer planning times. Furthermore, this separation may limit the planner's ability to exploit problem-specific optimizations that could otherwise reduce computational complexity. As a result, while the decoupled approach provides flexibility and re-usability, it can negatively impact the overall efficiency of the planning process. We build our encoding from the approach proposed in De Giacomo et al. (2017). In this part, we present a more explicit encoding that provides a clearer understanding of the planning problem. In contrast, in the *Java* tool in 4, we used a more compact encoding, more closely aligned with the original formulation (De Giacomo et al., 2017).
All the PDDL files shown in the following discussion can be found on our GitHub repository[1]. We proceed in the following way: we highlight each component step by step, gradually building up to the complete representation, leaving the overall syntax rules to the reader. We start with the **Planning Domain** $\mathcal{D}$ that models the execution of the automata that we discussed in the previous section. We describe the object types of our domain of interest, as illustrated in the accompanying figure.

[1]`https://github.com/GianmarcoBordin/LifecycleAligner`

```
(define (domain traceAlignment)
  (:types
        activity state - object
        automaton_state trace_state lifecycle_state - state
        lifecycle_activity - activity
  )
  ...
  (:constants
         init_state assigned_state started_state completed_state sink_state - lifecycle_sta
  )
  ...
```

We have two object types at the top level of the hierarchy: `activity` and `state`. The object type is the top-level hierarchy type in PDDL, representing the most general and fundamental type in the domain. Below are their descriptions:

- **activity**: This type represents tasks that can occur within the system. It serves as the parent for specific activities, such as `lifecycle_activity` (e.g, given a general activity a the specific life-cycle activities $a_{assign}, a_{start}, a_{complete}$).
- **state**: The `state` type refers to the different statuses that an automaton can be in (i.e, trace automaton, model and life-cycle automata). It is a parent type for specific states like `automaton_state`, `trace_state`, and `lifecycle_state`. These subtypes represent different kinds of states for each automaton that we can encounter:
  - `automaton_state`: Represents states in the model automaton.
  - `trace_state`: Refers to states related to the trace automaton.
  - `lifecycle_state`: Denotes the various states of the life-cycle automaton.

.

We note that, as shown in the picture, we defined constants. These are constant objects valid for any problem instance and they represent the life-cycle states of our life-cycle model 3.1.2. We provide a visual representation of the types hierarchy.

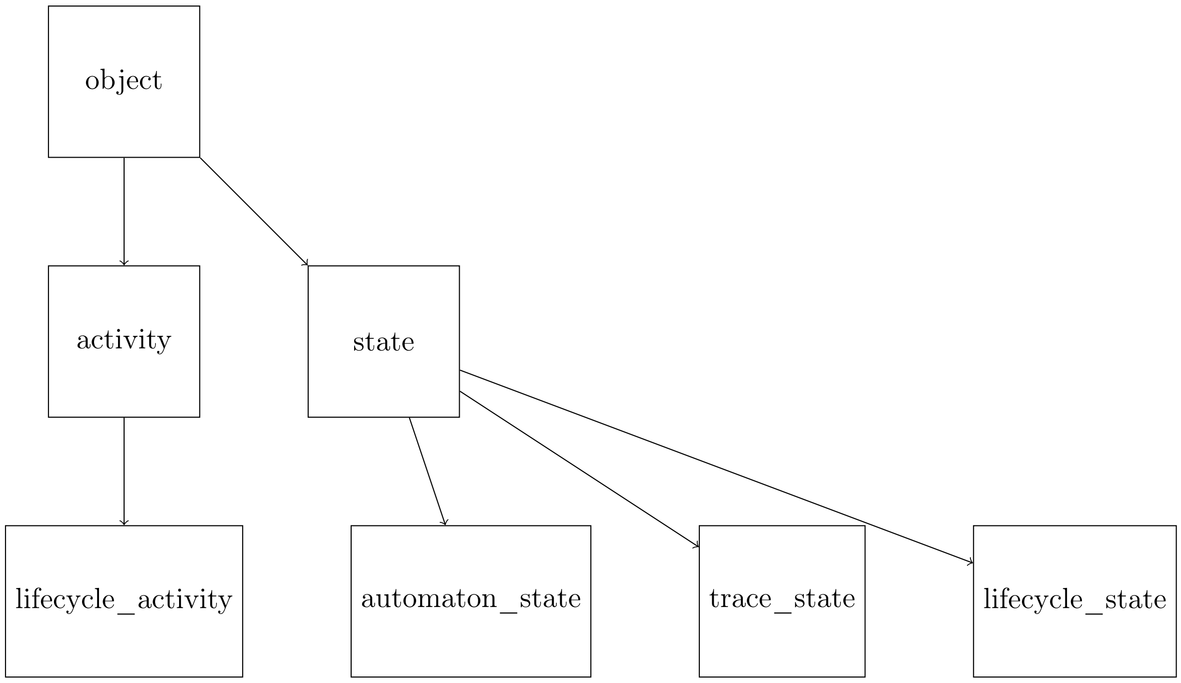


Figure 3.9: Example of a hierarchical structure of the object types.

We need now to define the boolean predicates that models the trace automaton, the model automaton and the life-cycle automaton.

```
  ...
(:predicates
    (trace ?t1 - trace_state ?l - lifecycle_activity ?t2 - trace_state)
    (automaton ?s1 - automaton_state ?l - lifecycle_activity ?s2 - automaton_state)
    (lifecycle ?l1 - lifecycle_state ?l - lifecycle_activity ?l2 - lifecycle_state)
    (lifecycle_activity_of ?a - activity ?l - lifecycle_activity)
    (cur_state ?s - state)
    (final_state ?s - state)
    (cur_lifecycle_state ?a - activity ?s - lifecycle_state)
    (final_lifecycle_state ?a - activity ?s - lifecycle_state)
)
  ...
```

As we can see from the above, we have the **cur-state** and **final-state** predicates used to model the initial and final accepting state of the automata. For the trace automaton the final state will be reached whenever we end to parse all the trace's activities/events. For the model automaton the accepting state is reached if all the constraints in the model are satisfied. Regarding our life-cycle automaton, we have two similar predicates, **cur-lifecycle-state** and

final-lifecycle-state that do the same. The difference between these predicates is that, for the former, it takes as parameter only a state object whilst for the latter, it needs also the general activity (e.g, a) that the life-cycle activity belongs to. This, will be changed in the following version but we reported this preliminary design to highlight better the changes with respect the original work of De Giacomo et al. (2017). The **trace** and **automaton** have the aim of simulating the transitions between states of the corresponding automaton, after encountering a life-cycle activity (e.g, a_{assign}). The newly created predicate **lifecycle** has the same aim but with respect to the life-cycle automaton. To identify the general activity a specific activity belongs to we employ the **lifecycle-activity-of** predicate, used to link the two objects.

We now discuss the actions that the planner can perform using our encoding. The first action, noted as **sync**, has the purpose of modeling the synchronous transitions between all the automata. This move will trigger transition in all the automata synchronously. The effect effectively tracks the current state changes in all automata. The preconditions always check that exists a certain transition in the involved automata and that we are currently in the state mentioned as parameters.

```
...
(:action sync
  :parameters (?t1 - trace_state ?l - lifecycle_activity ?t2 - trace_state)
  :precondition (and (cur_state ?t1) (trace ?t1 ?l ?t2))
  :effect (and (not (cur_state ?t1)) (cur_state ?t2)
            (forall (?s1 ?s2 - automaton_state)
                   (when (and (cur_state ?s1)
                              (automaton ?s1 ?l ?s2)
                         )
                         (and (not (cur_state ?s1))
                              (cur_state ?s2)
                         )
                   )
            )
            (forall (?a - activity)
                (forall (?l1 ?l2 - lifecycle_state)
                   (when (and (lifecycle_activity_of ?a ?l)
                              (cur_lifecycle_state ?a ?l1)
                              (lifecycle ?l1 ?l ?l2)
                         )
                         (and (not (cur_lifecycle_state ?a ?l1))
                         (cur_lifecycle_state ?a ?l2)
                         )
                   )
                )
            )
        )
)
...
```

We now describe only one of the three action, each is specific for a life-cycle transition (e.g, assign, start, complete). This formulation uses sink states like modeled in Life-Cycle model. If the **add-completed** is performed, but we are not in the started state (i.e, the expected one), we will reach the sink state of the life-cycle automata so never reaching our goal. For this reason the planner will not choose that path. This comes at a cost, because the planner has more states to visit. The effects are modeled similar to the previous action but now only the model and life-cycle automata are moving, for this reason it is referred to as *move in the model*.

```
...

(:action add_completed
       :parameters (?l - lifecycle_activity ?a - activity)
       :effect (and (increase (total-cost) 1)
                    (when (and (lifecycle_activity_of ?a ?l)
                               (cur_lifecycle_state ?a started_state)
                               (lifecycle started_state ?l completed_state)
                          )
                          (and (not (cur_lifecycle_state ?a started_state))
                            (cur_lifecycle_state ?a completed_state)
                          )
                    )
                    (forall (?s1 ?s2 - automaton_state)
                                  (when (and (cur_state ?s1)
                                             (automaton ?s1 ?l ?s2)
                                        )
                                         (and (not (cur_state ?s1))
                                            (cur_state ?s2)
                                         )
                                  )
                    )

              )
)
...
```

We provide an alternative definition that does not make use of sink states. This is obtained by only modifying the *when precondition* such that the model automaton does not move to the successor state if we are not in the started state. This leads the planner to explore less states with respect to the previous formulation, increasing the efficiency. It effectively blocks the model and life-cycle automata from moving to the next states.

```
...


(:action add_completed
       :parameters (?l - lifecycle_activity ?a - activity)
       :effect (and (increase (total-cost) 1)
                    (when (and (lifecycle_activity_of ?a ?l)
                               (cur_lifecycle_state ?a started_state)
                               (lifecycle started_state ?l completed_state)
                          )
                          (and (not (cur_lifecycle_state ?a started_state))
                            (cur_lifecycle_state ?a completed_state)
                          )
                    )
                    (forall (?s1 ?s2 - automaton_state)
                                  (when (and (cur_state ?s1)
                                             (automaton ?s1 ?l ?s2)
                                             (lifecycle_activity_of ?a ?l)
                                             (cur_lifecycle_state ?a started_state)
                                             (lifecycle started_state ?l
                                             completed_state)
                                           )
                                           (and (not (cur_state ?s1))
                                              (cur_state ?s2)
                                           )
                                  )
                    )


               )
)
...
```

The last action is a *move in the log*. We model the deletion of an action from the log. This makes only the trace automaton moving to the next state, leaving the other automata unchanged.

```
  ...

  (:action del
     :parameters (?t1 - trace_state ?l - lifecycle_activity ?t2 - trace_state)
     :precondition (and (cur_state ?t1) (trace ?t1 ?l ?t2))
     :effect (and (increase (total-cost) 1)
                  (not (cur_state ?t1)) (cur_state ?t2))
  )
  ...
```

We describe a simple sample problem instance for our domain. This instance has only a model constraint formula that in DECLARE is called *existence* constraint. It declares that the activity specified by the constraint must occur at least once in the trace. Our sample trace t is $t = a_{assign}, a_{complete}$.

```
(define (problem trace_Alignment_Problem.pddl)
        (:domain traceAlignment)

        (:objects
                t0 t1 t2  - trace_state
                s0 s1 - automaton_state
                b_assign b_start b_complete a_assign a_start a_complete - lifecycle_activit
        )
  ...
```

For the complete files we redirect to our GitHub repository that you can be found in LifeCycleAligner (2024). Our goal, as usually, is to align the trace with respect to the model and the life-cycle. To accomplish that we state that the current state must be equal to the final accepting state that we define in the init section of the problem for all the automata.

```
        (:goal
                (and
                        (cur_state s1)
                        (cur_lifecycle_state a completed_state)
                        (cur_state t2)
                )
        )
        (:metric minimize (total-cost))
)
```

A possible optimal plan for this problem instance is illustrated below.

Step	**Action**	**Cost**
0	(sync t0 a_assign t1)	0
1	(add_started a_start a)	1
2	(sync t1 a_complete t2)	0
3	(add_assign b_assign b)	1
4	(add_start b_start b)	1
5	(add_complete b_complete b)	1
Total Cost		**4**

Table 3.1: Plan with Actions and Associated Costs

We arrive to the end of this chapter, to conclude we only note that in the final tool implementation, to remain compliant with the original formulation (De Giacomo et al., 2017), we modified our PDDL encoding, considering the life-cycle constraints as *model constraints*. For the complete sources we redirect to our repository.

Chapter 4

Life-Cycle Aligner

After successfully defining the life-cycle model in Section 3.1.2, framing the life-cycle trace alignment problem as a planning problem, and encoding it in the PDDL language within a domain and corresponding planning problem, we now transition to the implementation phase of our approach. The discussion is divided in two fragments:

1. The *Not Grounded Life-Cycle Aligner* tool serves as an initial non-grounded implementation solution, leveraging the flexibility and simplicity of the *Python* programming language.
2. The *Grounded Life-Cycle Aligner* tool, a *Java*-based application designed to provide a grounded implementation, enhancing efficiency and performance.

All sources are available in our GitHub repository LifeCycleAligner (2024): it hosts the *Python* tool and the *Java*-based tool. During the discussion we also provide a running example of our approach to provide a better understanding of the mechanisms involved.

4.1 Tool Implementation Overview

To make a brief background on the file formats used during the following discussion, we provide a small introduction to *DOT* and *XES* (IEEE Task Force on Process Mining, 2023) standards. *DOT* is a plain text graph description language used by the *Graphviz* software[1] for creating visualizations of graphs. It provides a simple way to describe graphs using nodes and edges, usually employed to describe networks, we use it to represent automata. *XES* (eXtensible Event Stream) is a widely used XML-based format for storing event logs in process mining. It is designed to represent event logs from business processes, capturing event

[1]`https://graphviz.gitlab.io/`

attributes such as timestamps, activity names, and other contextual data, enabling analysts to study and mine business processes from raw data. The *PDDL* (Planning Domain Definition Language) Translator is a tool designed to generate lifecycle-constrained trace alignment PDDL problems from event logs in *XES* format and *DECLARE* models in *DOT* format. It leverages the *pm4py*[2] and *pydot*[3] libraries for this purpose. We use *PM4Py* to import and export event logs written in *XES* format, managing it as python data-structures, whilst we use *PyDot* to load *DOT* formatted model files into our script. The *DOT* file will contain our model automaton description (.dot) and the *XES* file will hold the event log traces (.xes). The translator outputs PDDL problem files compatible with the domain file, enabling planner-based trace alignment that accounts for the lifecycle of each activity in the log.
To summarize the overall tool functionalities:
The LifeCycleAligner (Not Grounded) is a python script that receives in input a model expressed in dot format and an event log given as xes format and outputs the pddl formatted problems that ensures life-cycle trace alignment for all the traces specified as parameters.
The domain and the problem specifications have already been presented in PDDL Encoding. We note that the sink-based formulation will be used for consistency reasons.

4.2 Non-Grounded Implementation

We begin with a formal definition for first-order languages. Let a first-order language be given, with:

- C: the set of constant symbols,
- F: the set of function symbols,
- P: the set of predicate symbols.

A **ground term** is a term that contains no variables. Ground terms can be defined recursively as follows:

1. Every element of C (the set of constant symbols) is a ground term.
2. If $f \in F$ is an n-ary function symbol and $\alpha_1, \alpha_2, \ldots, \alpha_n$ are ground terms, then

   $$f(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

   is a ground term.

[2]`https://pypi.org/project/pm4py/`
[3]`https://pypi.org/project/pydot/`

A **ground atom** (or **ground predicate** or **ground literal**) is an atomic formula where all argument terms are ground terms. Formally:

1. If $p \in P$ is an n-ary predicate symbol and $\alpha_1, \alpha_2, \ldots, \alpha_n$ are ground terms, then

$$p(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

   is a ground atom.

A **ground formula** (or **ground clause**) is a formula that does not contain any variables. Ground formulas can be defined recursively as follows:

1. A ground atom is a ground formula.

2. If φ and ψ are ground formulas, then:

   - $\neg\varphi$ (negation of φ) is a ground formula,
   - $\varphi \vee \psi$ (disjunction of φ and ψ) is a ground formula,
   - $\varphi \wedge \psi$ (conjunction of φ and ψ) is a ground formula.

To summarize, in our tool we output problems for our domain, where the actions, and consequently the preconditions and effects, are parameterized. The planner *has to substitute to this variables every possible objects that is compliant with the specified type.* We will see that this slows down the **searching** and **pre-processing time** of our state-of-the-art planner and for that reason we provide a more efficient tool in the second part of the treatment. With this tool, our aim, is to automatize the creation of the domain and problem formulation expressed in 3.3 and we do this based on the model input files and the xes event log.

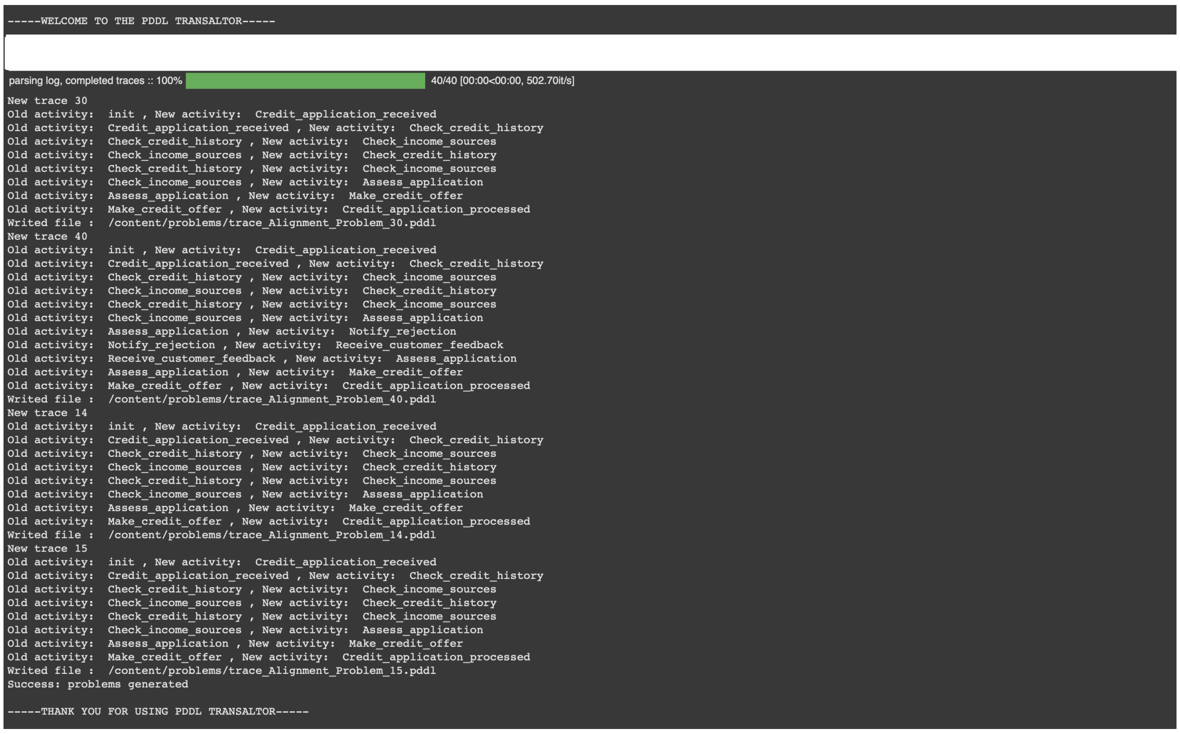
4.2.1 Interface and Execution

In this section we discuss the interface of our tool and we provide an explanation of the execution flow.
We explore the implementation provided in the .ipynb format, designed for enhanced portability and ease of execution. The log file in .xes is directly given through drive mount in the notebook, whilst the .dot file is asked to the user and it has to be chosen from the local folders. The tool then receives the input files and start constructing the problems instances. The parameters that the tool will use are the specified as global variables.

```
xes_file = "drive/MyDrive/TESI/SRC/simulation_logs.xes"
dot_files = ["declare1.dot"]
domain_name = "traceAlignment"
...
problem_automata_list = [1,2,5,9]
lifecycles = ["assign","start","complete"]
lifecycle_states =["init_state","assigned_state",
"started_state","completed_state","sink_state"]
...
activity_tag = "Activity"
lifecycle_tag = "lifecycle:transition"
... (other tags)
```

Figure 4.1: Sample tool parameters.

We point out that all these parameters can be customized by the user. The *xes-file* and *dot-files* variables track the input files location explained before. The domain-name specifies the pddl files prefix name. The *problem-automata-list* contains the traces number to parse (i.e, order of appearance in the log). The lifecycle parameters that follows, specifies the lifecycle model that the user want to follow with the related states, we could, for example, add other states in the life-cycle sequence. The last parameters are used to parse the event log and here it comes the *lifecycle-tag* that contains information about the activity lifecycle that we previously mentioned in 3.1.4.



```
-----WELCOME TO THE PDDL TRANSALTOR-----

parsing log, completed traces :: 100%  40/40 [00:00<00:00, 502.70it/s]
New trace 30
Old activity:  init , New activity:  Credit_application_received
Old activity:  Credit_application_received , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Assess_application
Old activity:  Assess_application , New activity:  Make_credit_offer
Old activity:  Make_credit_offer , New activity:  Credit_application_processed
Writed file :  /content/problems/trace_Alignment_Problem_30.pddl
New trace 40
Old activity:  init , New activity:  Credit_application_received
Old activity:  Credit_application_received , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Assess_application
Old activity:  Assess_application , New activity:  Notify_rejection
Old activity:  Notify_rejection , New activity:  Receive_customer_feedback
Old activity:  Receive_customer_feedback , New activity:  Assess_application
Old activity:  Assess_application , New activity:  Make_credit_offer
Old activity:  Make_credit_offer , New activity:  Credit_application_processed
Writed file :  /content/problems/trace_Alignment_Problem_40.pddl
New trace 14
Old activity:  init , New activity:  Credit_application_received
Old activity:  Credit_application_received , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Assess_application
Old activity:  Assess_application , New activity:  Make_credit_offer
Old activity:  Make_credit_offer , New activity:  Credit_application_processed
Writed file :  /content/problems/trace_Alignment_Problem_14.pddl
New trace 15
Old activity:  init , New activity:  Credit_application_received
Old activity:  Credit_application_received , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Check_credit_history
Old activity:  Check_credit_history , New activity:  Check_income_sources
Old activity:  Check_income_sources , New activity:  Assess_application
Old activity:  Assess_application , New activity:  Make_credit_offer
Old activity:  Make_credit_offer , New activity:  Credit_application_processed
Writed file :  /content/problems/trace_Alignment_Problem_15.pddl
Success: problems generated

-----THANK YOU FOR USING PDDL TRANSALTOR-----
```

Figure 4.2: Sample output for traces $\langle 30, 40, 14, 15 \rangle$

As we can see from the above, the selected traces are parsed and loaded in dictionary-like data structure in python. Our model in the .dot file is parsed with PyDot library to extract the model constraints.

```
digraph Automaton {
  rankdir = LR;
  initial [shape=plaintext, label=""];
  initial -> 0;

  0 [shape=circle, label="Start"];
  1 [shape=doublecircle, label="Accept"];

  0 -> 0 [label="!b_complete"];
  0 -> 1 [label="b_complete"];
  1 -> 1 [label="*"];
}
```

Figure 4.3: Sample `.dot` model file for existence(b-complete) declare constraint.

We point out that during our implementation we don't really track and consider self-loops, but only transitions that triggers state changes, for explicitness we reported it in these pictures.

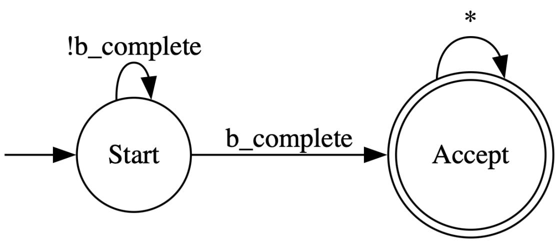


Figure 4.4: existence(b-complete) declare constraint Graphviz visualization.

The output of the tool is a directory of .pddl problem files that is automatically downloaded from the notebook to the local machine easing the local execution of one of the state-of the-art planning systems. During our discussion we will use the fast-downward planner (Helmert, 2006) and the symba-2* planner (Torralba et al., 2014), that is based on the former planner.

4.2.2 Algorithm and Implementation

In this subsection, our discussion is articulated in three core functions and we concentrate on each of them one at a time. We adopt a bottom-up approach, such that we start from the inner-most functions in the script and later we concentrate on the outer functions.

1. **write-pddl-problems**, used to write the problem files.
2. **generate-formula**, used to parse the dot and populate the related data structures to be used later.
3. **generate**, used to parse the xes and instantiate the principal data-structures. It also has the aim of calling the above mentioned functions.

1. **write-pddl-problems**. The function `write-pddl-problem` is designed to generate a PDDL problem file based on the parameters provided. It accepts the following inputs:

- `output_file (str)`: The name of the output file where the PDDL problem definition will be written.
- `problem_name (str)`: The name of the PDDL problem, used in the problem's definition.
- `initial_state (list)`: A list of strings representing the initial state of the PDDL problem, consisting of the predicates or facts that are true at the beginning.
- `objects (dict)`: A dictionary containing the objects in the PDDL problem, where the keys are object types, and the values are lists of object names.

- `goal_state (list)`: A list of strings representing the goal state of the PDDL problem, consisting of the desired predicates or facts that must be achieved.

The function outputs a PDDL-compliant problem file in the **directory** path variable, structured according to the specified parameters embedded as dynamic content and static content that is always written. Python string formatting handles the dynamic file content embedding.
2. **generate-formula**. The `generate-formula` function is responsible for populating the *transition* and *states* of the model automaton. It has to parse the nodes and edges extracted (as a graph structure) from the dot file thanks to the *PyDot* library. It firstly parses the state name labels so that it can extract the initial state and goal state of the model automaton. For all the states, it parses the edges labels of transitions between different states, ensuring the correct valorization of the automaton transitions, contained in the *transitions* key in the support data-structure *automaton.* The central part of the function has to create the life-cycle predicates needed for the life-cycle trace alignment in pddl. It does so by parsing each activity that it finds in the edge labels of the dot model. The life-cycle transitions are parameterized such that more states can be added. The procedure has the following steps:

- Each transition's label is analyzed to extract activity names and lifecycle constraints. The function performs the following:
    1. Identifies activities and their associated lifecycles.
    2. Adds lifecycle-related predicates for each activity.
- Updates `goal_state` to include the desired lifecycle states.
- Ensures that each activity is added to the `objects["activity"]` as a general activity and to the `objects["lifecycle_activity"]` lists as a specific lifecycle activity.

The function modifies the provided parameters (`initial_state`, `objects`, `goal_state`) in-place, ensuring they are fully populated with the PDDL elements derived from the *DECLARE* model. The function is designed to handle complex *DECLARE* models, including those with multiple lifecycles and nested model constraints, ensuring a robust and accurate translation into PDDL format.
3. **generate**. This function has the task of parsing each selected trace of the xes log. It performs the same computation used to populate the lifecycle predicates expressed before in the `generate-formula` function, that is, parsing each activity of each selected trace in the log and populating the related data-structures containing the life-cycle predicates formatted as strings. It also populate the adequate data structures for the trace automaton including

`initial_state`, `objects`, `goal_state`. It also parses the log and it also calls the generate formula to correctly update the current data-structures used in the current problem construction. For each problem it calls the above mentioned **write-pddl-problems** function to finalize the pddl file creation.

4.2.3 Running Example

After the creation fo the pddl domain and problems files, to compute the life-cycle trace alignment, we have to actually feed the planner with the pddl domain file and the pddl problem file for the specific trace of the event log. In this section we provide a simple running example that helps understanding the life-cycle alignment task afforded by the planner.
We use, as mentioned before, the *fast-downward* (Helmert, 2006) planning system.
To launch the life-cycle trace alignment, simply execute the following command:

```
python3 main.py
```

Upon running the script, the following output will be generated: **problem_name.pddl**: this file will be placed, together with the other problems for each selected trace in the event log, in the previously specified problems directory.
We use the $A*$ algorithm (Hart et al., 1968) to perform the search in combination with the blind heuristic (i.e, for each state n, $h(n) = 0$) that ensure us to find an optimal solution to the alignment problem. We also note that, to construct the .pddl files, we use the simple model defined in 4.3, obtaining the simple problem and domain instances already presented in 3.3. To perform lifecycle constrained trace alignment planning, we feed the planner with the *lifeCycleTraceAlignment* domain and problem files, using the following command:

```
./fast-downward.py {path/to/file/}{domain_name}.pddl {path/to/
    file/}{problem_name}_{problem_number}.pddl --search "astar(
    blind())"
```

Let us consider the optimal solution that we reported in 3.1. Following our example problem, the trace automaton has three states (i.e, t_0, t_1, t_2) whilst the model automaton has two states (i.e, s_0, s_1). We have one life-cycle automaton for each general activity (i.,e a, b) each with five states, tracking the life-cycle. We must correct the trace so the planner must satisfy the *DECLARE existence(b_complete)* constraint, adding the $b_{complete}$ activity to the trace, that only contains $\langle a_{assign}, a_{complete} \rangle$. In addition, it has to correct the lifecycle of both activities (i.,e a, b). We note that, the *del* action in this plan is not present, because it would cost more to delete two activities (i.e, $a_{assign}, a_{complete}$), in place of adding one a_{start}.

We now discuss each move step by step.

- **Step 0:** `(sync t0 a_assign t1)` The planner first employ a *sync* move for trace state t_0 with the event a_{assign}, transitioning to the next trace state t_1. The automaton does not enforce any conditions on this event.

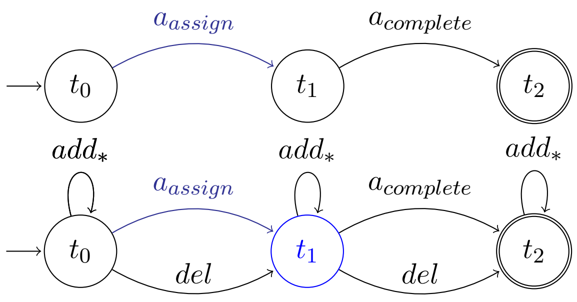
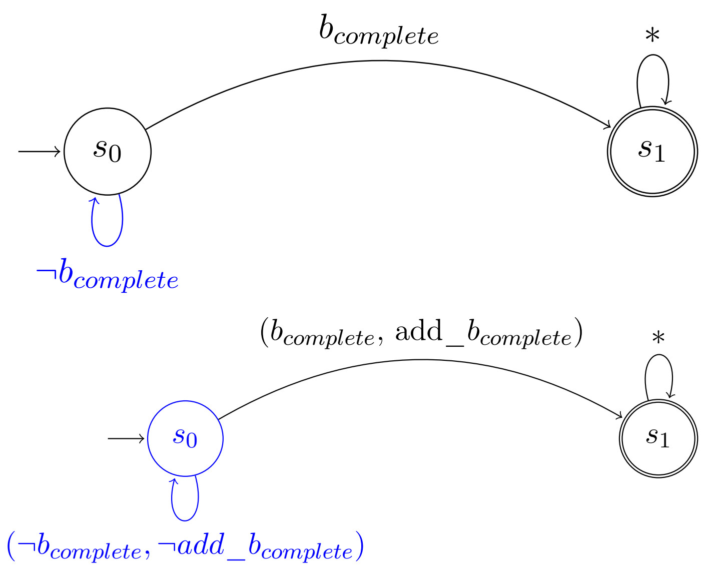


Figure 4.5: Step 0. Trace automaton and Augmented Trace automaton.



$b_{complete}$
$*$
s_0
s_1
$\neg b_{complete}$
$(b_{complete}, \text{add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.6: Step 0. Model automaton and Augmented Model automaton.

For the life-cycle automata we only report the augmented automata, to see the base version refers to 3.8. We note also that for visualization constraints we don't report the self-loops in the following automata, the complete automata design representation is left to the reader. Obviously the *a* lifecycle events are self-loops in *b* lifecycle automaton. This follows also our PDDL Encoding because we don't actually model self loops. We note that this is not a limitation because the automata current state information stays the same preventing any information loss.

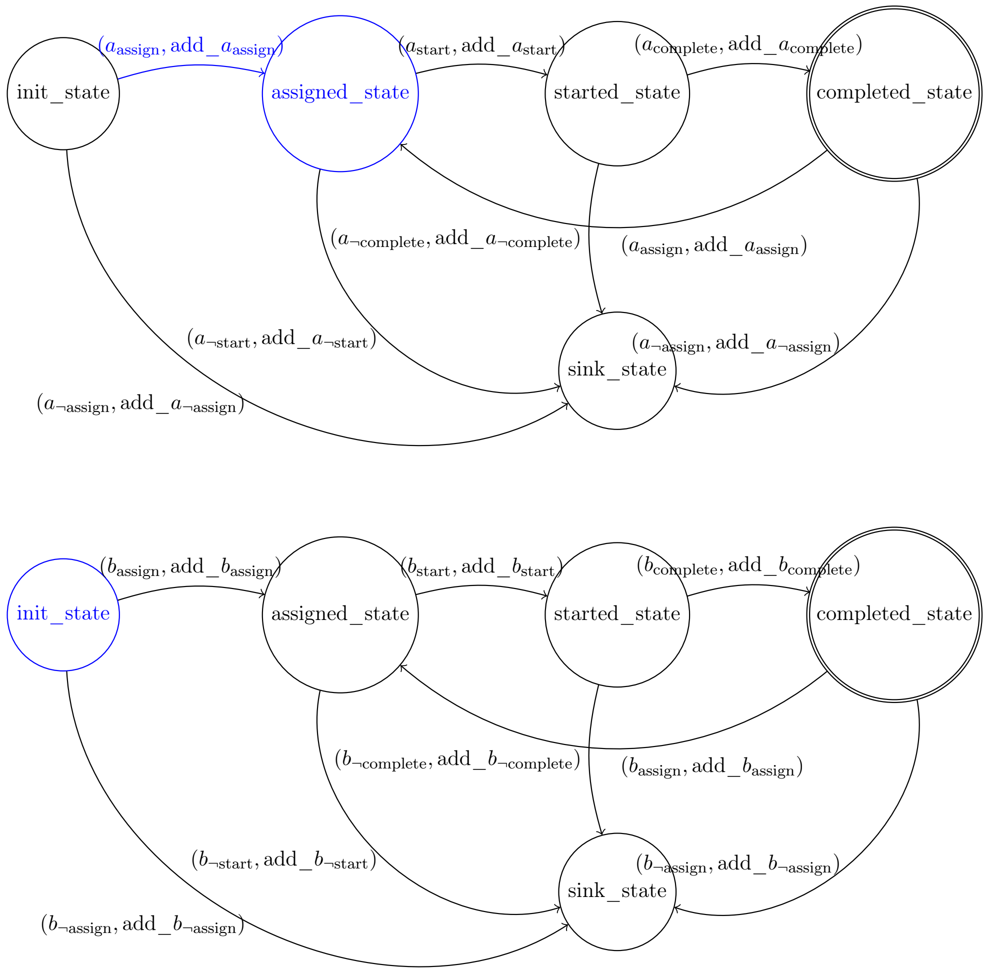

init_state
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
assigned_state
$(a_{\text{start}}, \text{add\_}a_{\text{start}})$
started_state
$(a_{\text{complete}}, \text{add\_}a_{\text{complete}})$
completed_state
$(a_{\neg\text{complete}}, \text{add\_}a_{\neg\text{complete}})$
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\neg\text{start}}, \text{add\_}a_{\neg\text{start}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
sink_state
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
init_state
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
assigned_state
$(b_{\text{start}}, \text{add\_}b_{\text{start}})$
started_state
$(b_{\text{complete}}, \text{add\_}b_{\text{complete}})$
completed_state
$(b_{\neg\text{complete}}, \text{add\_}b_{\neg\text{complete}})$
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\neg\text{start}}, \text{add\_}b_{\neg\text{start}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$
sink_state
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$

Figure 4.7: Step 0. Augmented lifecycle automata for activities a and b.

Where for $b_{\neg\text{assign}}$, we mean $\langle b_{\text{start}}, b_{\text{complete}} \rangle$, and similarly for other labels such as $a_{\neg\text{assign}}$, which corresponds to $\langle a_{\text{start}}, a_{\text{complete}} \rangle$, and so forth.

- **Step 1:** `(add_started a_start a)` At this point, the planner could employ an *add_started* action to progress forward in the a-life cycle automaton. However, the trace automaton does not transition to the next state, nor does the model automaton. As a result, the b-life cycle automaton also remains unchanged.

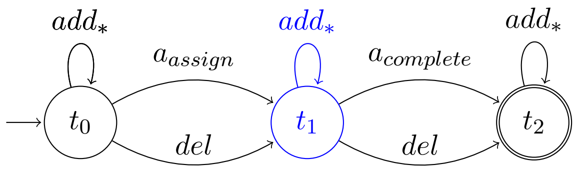
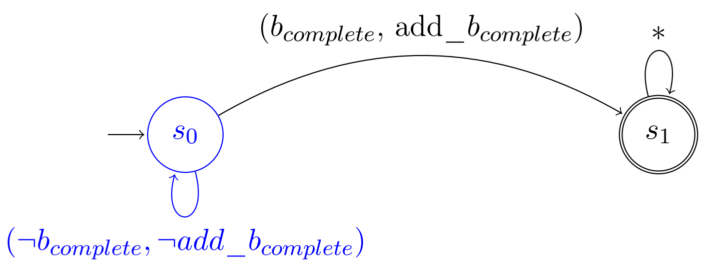
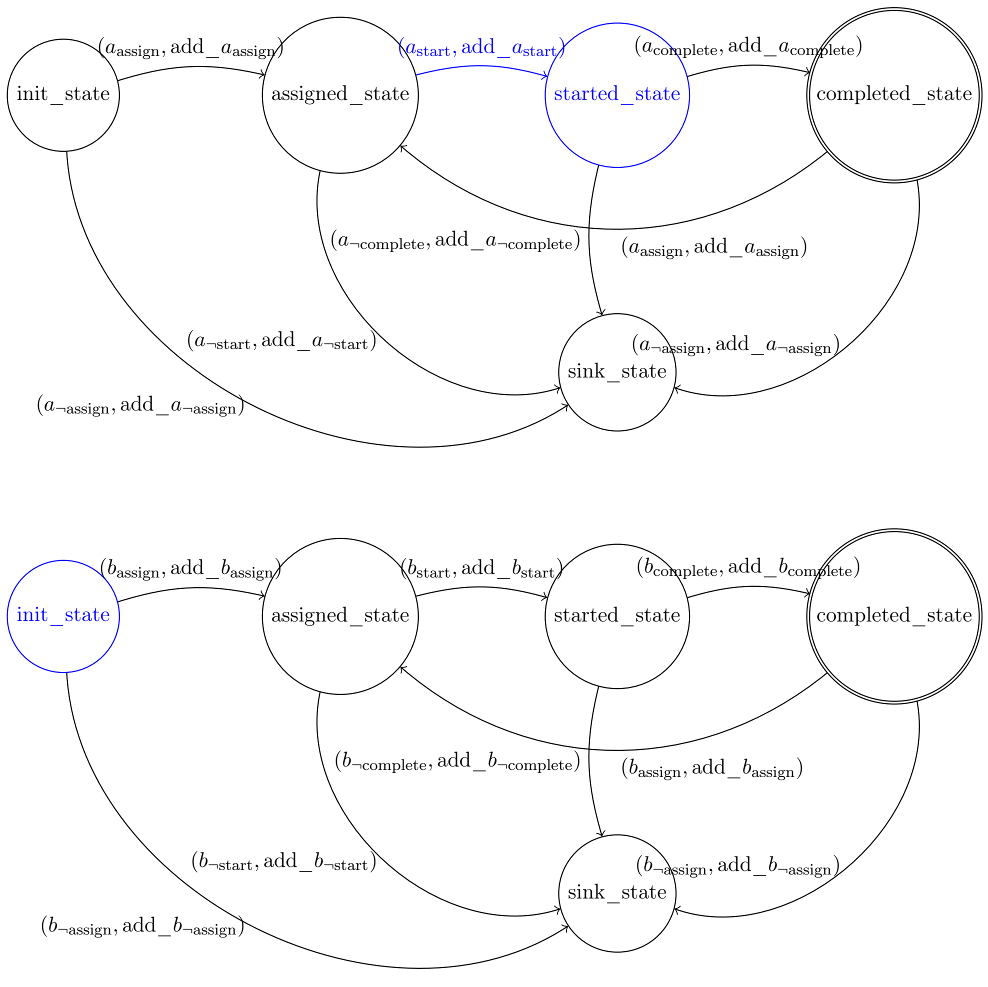


Figure 4.8: Step 1. Augmented Trace automaton.



$(b_{complete}, \text{add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.9: Step 1. Augmented Model automaton.



init_state
assigned_state
started_state
completed_state
sink_state
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\text{start}}, \text{add\_}a_{\text{start}})$
$(a_{\text{complete}}, \text{add\_}a_{\text{complete}})$
$(a_{\neg\text{complete}}, \text{add\_}a_{\neg\text{complete}})$
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\neg\text{start}}, \text{add\_}a_{\neg\text{start}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
init_state
assigned_state
started_state
completed_state
sink_state
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\text{start}}, \text{add\_}b_{\text{start}})$
$(b_{\text{complete}}, \text{add\_}b_{\text{complete}})$
$(b_{\neg\text{complete}}, \text{add\_}b_{\neg\text{complete}})$
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\neg\text{start}}, \text{add\_}b_{\neg\text{start}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$

Figure 4.10: Step 1. Augmented lifecycle automata for activities a and b.

- **Step 2:** `(sync t1 a_complete t2)` The planner now uses a *sync* move to complete the life-cycle of a. The trace automaton goes to the accepting state because the trace has been completely parsed. The other automaton are unchanged except for the a lifecycle automaton that reaches the final accepting state.

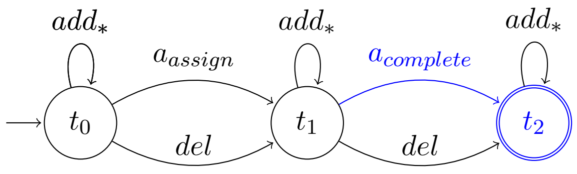


Figure 4.11: Step 2. Augmented Trace automaton.

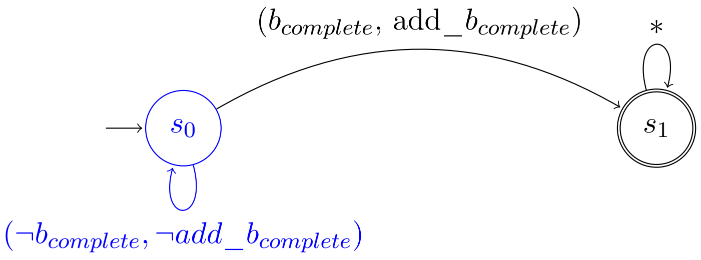

$(b_{complete}, \text{add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.12: Step 2. Augmented Model automaton.

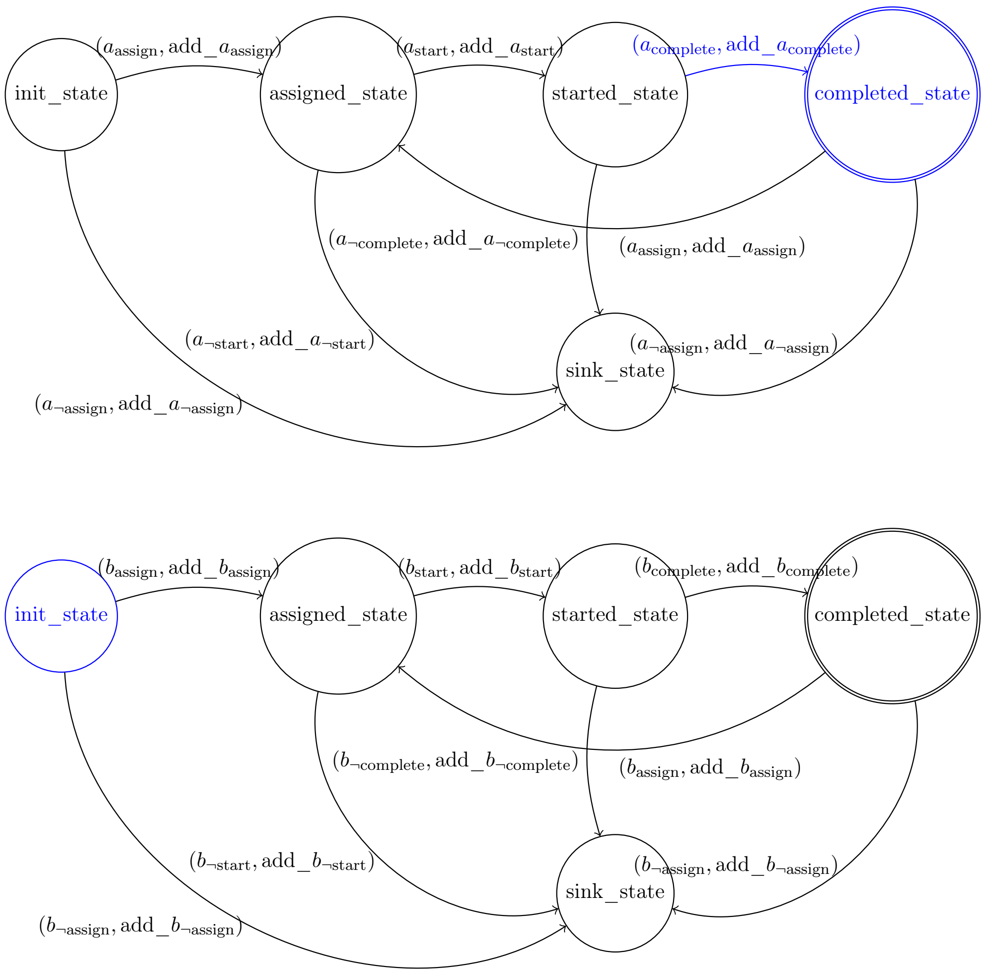

init_state
assigned_state
started_state
completed_state
sink_state
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\text{start}}, \text{add\_}a_{\text{start}})$
$(a_{\text{complete}}, \text{add\_}a_{\text{complete}})$
$(a_{\neg\text{complete}}, \text{add\_}a_{\neg\text{complete}})$
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\neg\text{start}}, \text{add\_}a_{\neg\text{start}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
init_state
assigned_state
started_state
completed_state
sink_state
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\text{start}}, \text{add\_}b_{\text{start}})$
$(b_{\text{complete}}, \text{add\_}b_{\text{complete}})$
$(b_{\neg\text{complete}}, \text{add\_}b_{\neg\text{complete}})$
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\neg\text{start}}, \text{add\_}b_{\neg\text{start}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$

Figure 4.13: Step 2. Augmented lifecycle automata for activities a and b.

- **Step 3:** `(add_assign b_assign b)` At this step, the planner has to satisfy the model existence constraint. But for doing that it has also to correct its life-cycle and it adopts an add move. The other automata don't change.

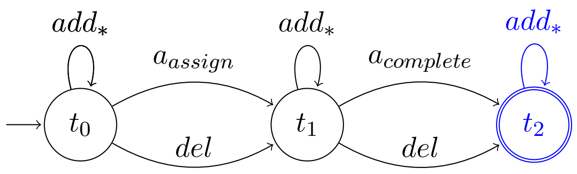


Figure 4.14: Step 3. Augmented Trace automaton.

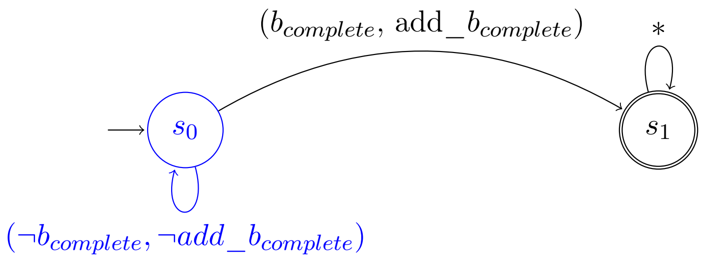

$(b_{complete},\ \text{add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.15: Step 3. Augmented Model automaton.

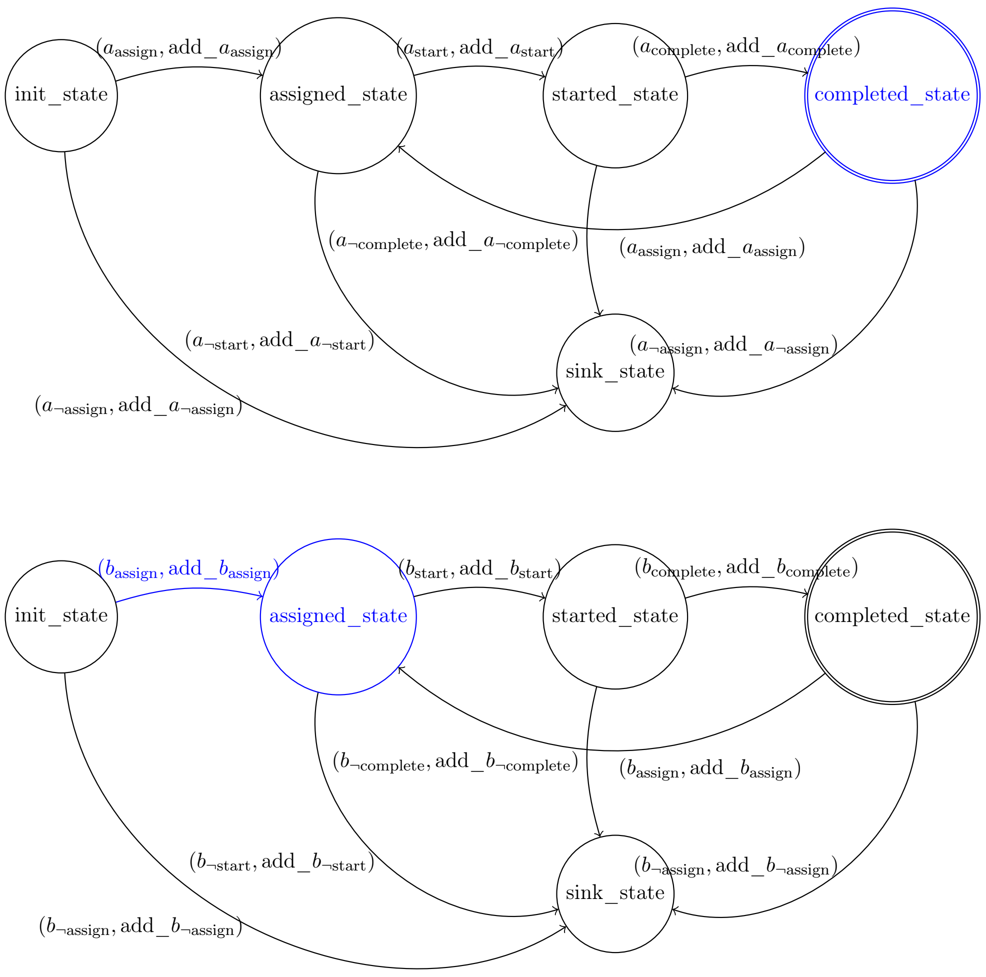

init_state
assigned_state
started_state
completed_state
sink_state
$(a_{assign}, add\_a_{assign})$
$(a_{start}, add\_a_{start})$
$(a_{complete}, add\_a_{complete})$
$(a_{\neg complete}, add\_a_{\neg complete})$
$(a_{assign}, add\_a_{assign})$
$(a_{\neg start}, add\_a_{\neg start})$
$(a_{\neg assign}, add\_a_{\neg assign})$
$(a_{\neg assign}, add\_a_{\neg assign})$
init_state
assigned_state
started_state
completed_state
sink_state
$(b_{assign}, add\_b_{assign})$
$(b_{start}, add\_b_{start})$
$(b_{complete}, add\_b_{complete})$
$(b_{\neg complete}, add\_b_{\neg complete})$
$(b_{assign}, add\_b_{assign})$
$(b_{\neg start}, add\_b_{\neg start})$
$(b_{\neg assign}, add\_b_{\neg assign})$
$(b_{\neg assign}, add\_b_{\neg assign})$

Figure 4.16: step 3. Augmented lifecycle automata for activities a and b.

- **Step 4:** `(add_start b_start b)` The planner continues correcting the life-cycle of the newly added activity in the trace (i.,e b). The other automata are left unchanged.

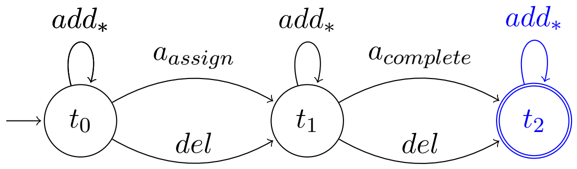
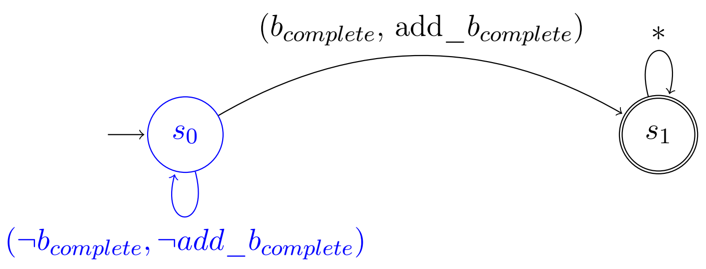
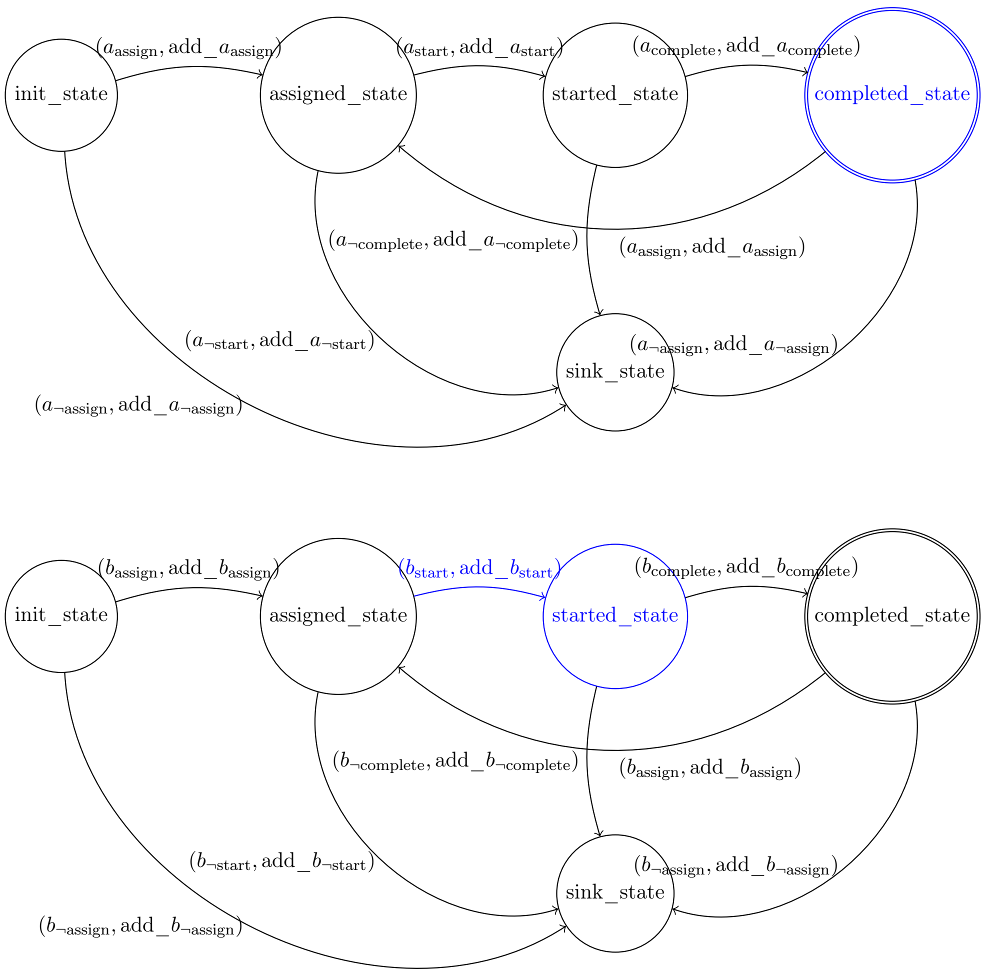


Figure 4.17: Step 4. Augmented Trace automaton.



$(b_{complete}, \text{add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.18: Step 4. Augmented Model automaton.



init_state
assigned_state
started_state
completed_state
sink_state
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\text{start}}, \text{add\_}a_{\text{start}})$
$(a_{\text{complete}}, \text{add\_}a_{\text{complete}})$
$(a_{\neg\text{complete}}, \text{add\_}a_{\neg\text{complete}})$
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\neg\text{start}}, \text{add\_}a_{\neg\text{start}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
init_state
assigned_state
started_state
completed_state
sink_state
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\text{start}}, \text{add\_}b_{\text{start}})$
$(b_{\text{complete}}, \text{add\_}b_{\text{complete}})$
$(b_{\neg\text{complete}}, \text{add\_}b_{\neg\text{complete}})$
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\neg\text{start}}, \text{add\_}b_{\neg\text{start}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$

Figure 4.19: Step 4. Augmented lifecycle automata for activities a and b.

- **Step 5:** `(add_complete b_complete b)` The planner finalizes the life-cycle correction of b and, at the same time, satisfies the *existence(b_ complete)* constraint.

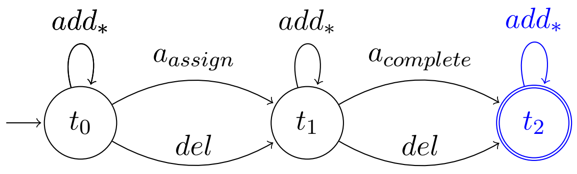
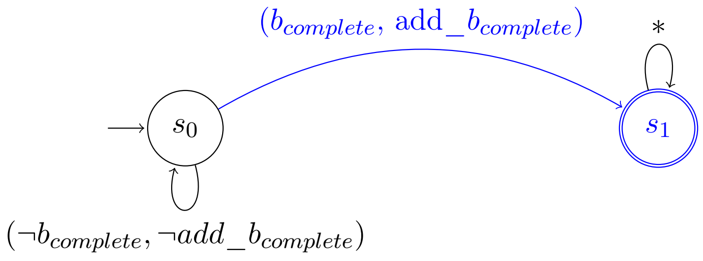
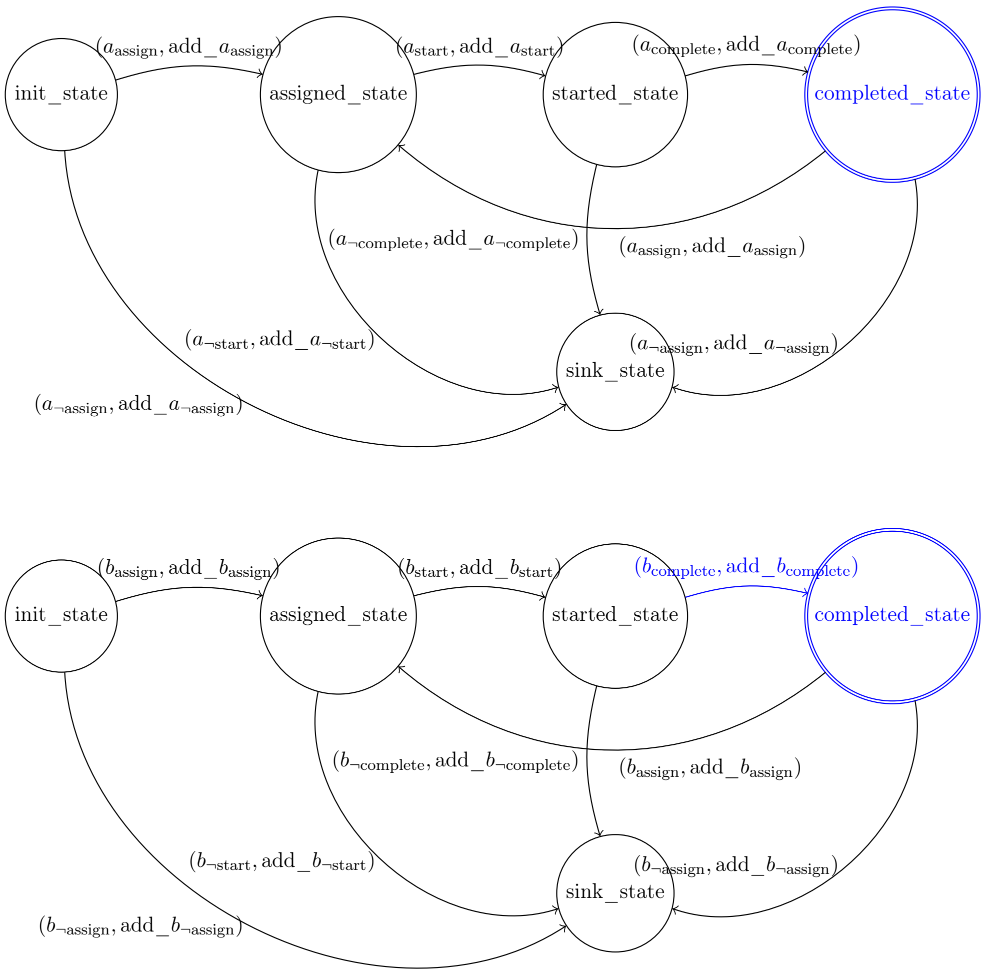


Figure 4.20: Step 5. Augmented Trace automaton.



$(b_{complete}, \text{ add\_}b_{complete})$
$*$
s_0
s_1
$(\neg b_{complete}, \neg add\_b_{complete})$

Figure 4.21: Step 5. Augmented Model automaton.



init_state
assigned_state
started_state
completed_state
sink_state
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\text{start}}, \text{add\_}a_{\text{start}})$
$(a_{\text{complete}}, \text{add\_}a_{\text{complete}})$
$(a_{\neg\text{complete}}, \text{add\_}a_{\neg\text{complete}})$
$(a_{\text{assign}}, \text{add\_}a_{\text{assign}})$
$(a_{\neg\text{start}}, \text{add\_}a_{\neg\text{start}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
$(a_{\neg\text{assign}}, \text{add\_}a_{\neg\text{assign}})$
init_state
assigned_state
started_state
completed_state
sink_state
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\text{start}}, \text{add\_}b_{\text{start}})$
$(b_{\text{complete}}, \text{add\_}b_{\text{complete}})$
$(b_{\neg\text{complete}}, \text{add\_}b_{\neg\text{complete}})$
$(b_{\text{assign}}, \text{add\_}b_{\text{assign}})$
$(b_{\neg\text{start}}, \text{add\_}b_{\neg\text{start}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$
$(b_{\neg\text{assign}}, \text{add\_}b_{\neg\text{assign}})$

Figure 4.22: Step 5. Augmented lifecycle automata for activities a and b.

The model automaton reaches the final accepting state, and so does the b-life-cycle automaton. All the automata have reached their respective accepting states, thus achieving the goal. The cost of the optimal found plan is 4.

4.3 Grounded Implementation

The last implementation presented has the advantage that is very easy to comprehend but the disadvantage that the *add,del,sync* actions, that the planner can use, are not grounded. This means that, when the planner has to take a decision on the current action to perform, on a given state, it has to substitute to the action variables all the admissible combinations of instances. This can lead to the famous state explosion problem, where exploring all the interested states is computationally intractable. In addition many actions instances does not lead us closer to the solution, even more some action instances could contain inconsistent or erroneous parameters combinations.

This problem take us to adopt a more advanced approach to reach our final solution. In this section we present our method to cope with the previously listed problems using our tool, the *Life-Cycle Aligner* (Grounded) implementation. The tool creates $\langle domain_i.pddl, problem_i.pddl\rangle$, meaning that the domain and the problem instance are not independent from each other. In fact, the program creates strictly linked instances that allows the planner to efficiently search in the state space, reducing the overall number of nodes to search for. This approach contrasts with the original *PDDL* methodology but enables finding a solution in significantly shorter times. The tool also utilizes a novel mechanism to embed life-cycle enforcement directly into the .pddl problems. We illustrate the grounding algorithm, after that, we present the interface and its architecture, providing also the implementation details.

4.3.1 Algorithm

- 1. **Trace Automaton Creation**. At the beginning, the tool receives an event log $\mathcal{L}$ and a constraint model $\mathcal{M}$. Subsequently, it identifies all the events present in the log, extracting the *lifecycle:transition* and *concept:name* information to encode the specific life-cycle activity and tracking it in its trace alphabet $\Sigma^t_{\mathcal{L}}$ with t the trace involved, constructing the so called *trace automaton* as already illustrated in 3.1.1.

- 2. **LifeCycle And Model Automata Creation**. We are going to describe the life-cycle embedding, pointing out that there are several methods already implemented in our tool that leads to the same outcome and we will discuss it later in 4.3.2.

Our main method constructs Deterministic Finite Automaton (DFA) to specify the life-cycle of the general activities computed from Σ (e.g, during the parsing of the log we also track the general activities that occurs). When a **new** general activity a is discovered from the log, we create its DFA following the model described in 3.1.2. We note it as $\mathcal{A}_{life-cycle}$ and we call it *life-cycle automaton*. It will be treated, in the grounding, as a constraint automaton. Completed the parsing of all the log traces, it analyzes the constraint model, populating the constraints alphabet $\Sigma_{\mathcal{N}}$. The union of the two alphabet, namely $\Sigma = \Sigma^t_{\mathcal{L}} \cup \Sigma_{\mathcal{N}}$ forms the alphabet for a trace t (i.e, the specific problem instance). Moreover, for each constraint, in $\mathcal{M}$ we construct a DFA, called *constraint automaton*.

- 3. **Add Actions Creation**. The steps described in this segment are valid for the creation of *add* actions.
  - 3.1 **Augmenting Phase**. At this point the *grounding* starts and we want the automata to be minimal, in the sense that, we identify all the paths in the automata that don't lead to the achievement of the goal and we erase those states from the automata. Now, starting from the reduced automata, we create a synthetic state named *abstract* that is an accepting state and is connected to any accepting state of each automata through appropriate special transitions. We augment each automaton like we explained in 4.2.3.
  - 3.2 **Relevant Transition Phase**. Subsequently, we identify all the **relevant transitions** of each $\mathcal{A}^+$ and $\mathcal{A}^+_{life-cycle}$. That is, all transitions between different states. We create labels that help to identify the specific transitions. Next, based on their labels, we group the relevant transitions using the specific activity which it refers to (e.g, a_{assign}). For each group, we have to find all simple combinations without repetition of length k, where $1 \leq k \leq M$ and M distinct automata cardinality contributing to at least one transition in the set. We note that, many combinations containing transitions of the same automaton will not be considered because only one transition at a time can occur in an automaton. Moreover, all the combinations containing transitions belonging to the same automaton or that have been dropped in the minimal automaton, must be skipped.
  - 3.3 **Action Generation Phase**. For *preconditions*, the source states of the transitions in the combination will be evaluated. The action will be executed only if the current state is not among the source states of the transitions that are part of

the group referenced by the combination but are excluded from it. The *effects* will apply exclusively to transitions between different states, targeting the destination states of the transitions in the combination and the negation (¬) of their source states. The *cost* of the action will be the total sum of the costs of all transitions in the combination.

- 4. **Del Actions Creation**. The procedure to generate *del* actions is the straightforward as we only need to generate actions to move from one trace state t_i to t_{i+1}.
- 5. **Sync Actions Creation**.
  - 5.1 **Sync Actions Type 1** For this type, it is necessary to generate a *sync* action for each *add* action that we provided before.
  - 5.2 **Sync Actions Type 2** It is also necessary to permit loop transitions, when no outgoing transitions exist in a specific state of any automaton, for each label in the trace.
  - 5.3 **Sync Actions Type 3** Is is needed to generate also *sync* actions for labels in the trace but not in the model automata.
- 6. **Planning Domain And Problem Generation**. Finally, we have to generate the planning domain. The encoding is based on the one described in De Giacomo et al. (2017), including *state* object types and the related predicates, using the newly created actions. For the planning problem, we generate the relative constants objects related to the specific trace, the model and life-cycle automata. After that, we initialize the current states of all automata and we define our goal, that is, **the conjunction of the trace automaton and all the constraint automata final states**. By using the abstract states obtained before for the constraint and life-cycle automata, we avoid disjunctions in goal formulas, which are not supported by all planners. Our metric will be minimize the total-cost.

4.3.2 Architecture and Implementation

In this subsection, we describe the main components and classes of our *Java* tool, focusing on the most important implementation sections. The `main` class will be discussed in the CLI and Execution subsection, so we focus on the other modules. To start with the algorithm implementation previously discussed, we have to load the specific activities to populate the data-structures used during the execution.

- This task is in charge of the `Loader` class, that has the following methods. `loadFile` is a general method to switch to the specific file method. It handles the alphabet instantiation and the data-structure valorization. In the case of constraints file methods, like `LoadXml` and `LoadDot`, we also load to the `Container` class the constraints that we will need during the previously discussed algorithm, one at a time.

- The `Container` class is left to the reader, due to its straightforward intuitiveness. It has the aim of hosting the overall data-structures (e.g, alphabet list, constraint list ...) and other general constants used by the other modules.

- The `LifeCycle` class is in charge of handling life-cycle enforcement for each activity a. This aim can be achieved using different variants.
  We can embed declare constraints to specify the lifecycle, appending it to the .xml declare model file given as input in the tool execution (through the `combine` method where the user can switch between the two default life-cycle types based on a flag settled in the `Container` class) or we can create .dot files that specify this lifecycle (through the `createLifecycleDotChain` and `createLifecycleDotAlternateChain` functions). In the tool we defined two different lifecycle variants. In the *chain-succession* variant, we employ two chain-succession constraints ($\langle chainsuccession(assign, start), chainsuccession(start, complete)\rangle$). In the *alternate chain-succession* case, we use three declare alternate-chain-succession constraints ($\langle alternatesuccession(assign, start), alternatesuccession(start, complete), alternatesuccession(assign, complete)\rangle$).

- The `Utilities` class is the next module encountered during the execution. The core functions are contained in it. The `preprocessLifecycleDot` method adds flexibility to the customization of the life-cycle enforcing. To enhance flexibility, the user can modify the dot template named *lifecycle.dot*. The `preprocessLifecycleDot` function is responsible for inserting the activity name into the general template, ensuring that the life-cycle's integrity, for each activity, is maintained. This ensures the correct life-cycle progression is enforced across various types of lifecycles, enhancing our former model based on a simple sequence of life-cycle states (i.e, $\langle assign, start, complete\rangle$). This modification blurs the boundary between life-cycle constraint and model constraint (remember that the model constraints are only defined on *complete* tagged specific activities). Obviously, the performances of the tool varies depending on the specific life-cycle model to enforce. The method creates a custom life-cycle file for each activity following the interface of the life-cycle functions described before. The `getAutomatonForModelLearning` method has to transform the automata-based model into an automaton data structure composed of edges and nodes. This will be used in the `Generator` class, during the constraint parsing. The `findCombinationsOfTransitions` and `findCombinationsOfStates` functions handle the grounding algorithm step of finding all the relevant transitions and states of any automata encountered. This method recursively generates combinations of transitions and states from an array. It ensures that no two transitions from the same automaton are selected together. The process works as follows:

  - **Base Case** (`len` == 0):
    The method checks if the combination contains transitions from the same automaton. If so, it discards the combination; otherwise, the combination is added to the container as a valid combination.
  - **Recursive Case**:
    The method iterates through the array starting from `startPosition` and constructs combinations by recursively reducing `len`.

This functions are recursive and constitute the main inevitable bottleneck of the tool, needed to speed-up the planning search. The last two functions, `createPropositionalDomain` and `createPropositionalProblem`, handle the instantiation and valorization of the data-structures used to write the planner files, concatenating static and dynamic content given by the `Container` class.

- The `Generator` class, is the next module encountered during the execution. To summarize the class responsibility, it is in charge of calling the utilities functions already described to implement the grounding algorithm discussed. It parses all the constraints contained in the `ConstraintsListModel` of the `Container` class and for each of them:

  1. Generates the corresponding automata
     through the `getAutomatonForModelLearning` function.
  2. Finds the relevant transitions of the automata through
     `findCombinationsOfTransitions` and `findCombinationsOfStates` functions.
  3. Populates the required data-structures to build the planner files.
  4. Calls the `createPropositionalDomain` and
     `createPropositionalProblem` functions and creates the pddl files.

- The `Runner` class is responsible for executing the planner's bash scripts with the corresponding command-line arguments, allowing for seamless switching between the selected planner, as specified by the command-line input during the tool's execution.

4.3.3 CLI and Execution

As in the previous section, we examine the tool's interface provided as a Command-Line Interface (CLI) and outline its execution flow.
The tool interface is similar to the one provided in the previous section. The tool takes as input:

- The planner name.
- A .xes file representing the event log.
- A .xml model constraints file.

We point out that the tool can handle also .dot constraints files. Furthermore, the tool can take as input a generic list of model files formatted as .dot or .xes. The planner name parameter serves as a toggle to select between the two planners used in our tests: *fast-downward* and *symba2*, despite this, the tool can be easily modified to address any other state of the art planner, only requiring little effort.

```java
    public static void main(String[] args) throws Exception {

        long startTime;
        long endTime;
        long duration;
        String num_constraints = "";
        String noise_percentage = "";
        String avg_trace_length = "";
        String lifecycle = "";
        String planner;

        // Validate and Parse Command-Line Arguments
        ...

        Loader.loadFile(file, fileExtension);

        debug("----- GROUNDING PHASE STARTED -----");

        if (!Generator.goToPlanner()) {
            debug("Something during generation of pddl files went
               wrong");
            System.exit(-1);
        }

        debug("----- PDDL FILES GENERATION PHASE STARTED -----");

        Generator.generatePddlFiles();

        debug("----- RESULT PHASE STARTED -----");

        if (Container.getFDOptimalCheckBox()){
            (Runner.invokePlannerFast(duration,num_constraints,
               noise_percentage,avg_trace_length,lifecycle);
        } else if (Container.getSymBAoptimalCheckBox()) {
            (Runner.invokePlannerSymba(duration,num_constraints,
               noise_percentage,avg_trace_length,lifecycle);
        }
        // Backing up files
        ...
    }
```

In the above picture we can see the *main* method of our *Main* class. At the beginning, we declare many variables that we will use to correctly dump the pddl files for the specific execution with the related parameters and the help of the *Utilities* class. The function parses each input file, extracts the dump information, instantiates and populates the support data structure used in subsequent computations via the *Loader* class. Then the *Generator* class has to generate the pddl domain and problem files couples for each trace in the .xes log. The last module called is the *Runner* class. Its aim is to feed the selected planner with each $\langle domain_i.pddl, problem_i.pddl \rangle$ tuple to finalize the *life-cycle-aware trace alignment*.

Chapter 5

Testing and Evaluation

In this chapter, we describe the overall performances of our *LifeCycleAligner* (Grounded) tool. In the first section 5.1, we introduce the **noise** concept along with an ad-hoc created *Python* script named *Noiser.py*, which is used to introduce noise into our logs. Later in 5.2, we present our experimental setup, including hardware, software and datasets used. In the last part, we concentrate on the Evaluation of the Results, where we present our results and summarize key insights derived from them.

5.1 Noise Injection

In the *trace alignment* context, **noise** plays a critical role in evaluating alignment between system traces and predefined behavioral models. Noise quantifies the extent to which the observed system behavior (i.e, the event log evaluated) deviates from these expectations. In other words, noise quantifies the degree of corruption in a trace t, representing the extent to which the information within the trace deviates from or is inconsistent with the expected correct behavior. In our experiments, we consider the noise only with respect to the model constraints (e.g, so only on complete-tagged activities $a_{complete}, b_{complete}, \dots$).

Definition 21 *Let $\mathcal{M}$ be the constraints model, given a system trace t of length n, the noise $\eta(t, \mathcal{M})$ is defined as the total number of constraint violations (with respect to a model $\mathcal{M}$) per unit length of the trace. Formally, if $c = \{c_1, c_2, \ldots, c_m\}$ represents a set of behavioral constraints applied to the trace, then the noise $\eta(t, \mathcal{M})$ is defined as:*

$$\eta(t, \mathcal{M}) = \frac{1}{n} \sum_{i=1}^{m} V(c_i, t)$$

where:

- *n is the length of the trace t.*
- *$V(c_i, t)$ is a function that counts the number of times the constraint c_i, belonging to the model $\mathcal{M}$, is violated in the trace t.*

The noise $\eta(t, \mathcal{M})$ quantifies the normalized number of instances in which any of the given constraints c are violated throughout the trace. Consequently the *noise percentage* represent the above defined value, scaled to a 0–100% range.

Having defined noise, we need to introduce it into our experimental event logs to evaluate its impact. To accomplish that, we developed an ad-hoc tool called `Noiser`.

The `Noiser.py`, is a *Python*-based script that: takes as input an event log and for each trace of the log (that we import with the usual libraries in a dictionary-like data-structure), it injects the settled noise percentage value. It does that following an embedded model, formalized as usual with declare constraints. We focus on the models and other test parameters in the 5.2.

Now we give an overview of our tool, pointing out that it has the additional functionality to format our logs with respect the parsing libraries used in the *LifeCycleAligner* tool. The tool injects noise through modifications of the trace t in input with the help of the following moves:

- *Add* move, that is used to add an activity in the log trace t.
- *Del* move, that is used to delete an activity in the log trace t.
- *Swap* move, that is used to swap two activities in the log trace t.

Firstly, the tool calculates the number of total deviations to apply to the trace t. Secondly, it computes the amount of *Add*, *Del* and *Swap* moves to apply.

Thirdly, it parses the formatted trace t of the log and calls the `inject_noise` function that *while-loops* until the required number of deviations have been applied. We also note that we applied some tricks to make the tool work: if in a trace t, for a specific combination of

activities, the *del* or *swap* moves cannot be applied (e.g, one or more activities to delete or swap are not present in that trace t), the tool handles the corruption with *add* type moves because they can always be applied.
In this loop, we sample from the possible *add,del,swap* constraints (i.e, we identify a constraint as *add,del,swap* if in the tool its deviation is implemented with the corresponding move type) and we apply the implemented move to violate it through specific implementations of *add,del,swap*. The functions are straightforward to comprehend and we provide some of them in the following picture. At the end, as usual we export the corrupted logs.
We point out that the employed declare constraints for generic a and b activities are:

- *Existence(a)*, to violate it we remove all occurrences of a from the trace t.
- *Absence(a)*, to violate it we add an occurrence of a to the trace t.
- *Response(a,b)*, to violate it we remove all occurrences of b after a from the trace t.
- *NotSuccession(a,b)*, to violate it we swap a and b in the trace t.
- *NotChainSuccession(a,b)*, to violate it we add an occurrence of b next to a to the trace t.

```
def violate_response(trace, event, event2):
    if not act_in_trace(trace, [event,event2]):
        return -1
    new_trace = []
    event_found = False

    for e in trace:
        if event_found:
            if e[event_id_tag] == event2 and e[lifecycle_tag] ==
                "complete":
                continue
        elif e[event_id_tag] == event and e[lifecycle_tag] == "
            complete":
            event_found = True

        new_trace.append(e)
    trace[:] = new_trace
```

```
def violate_chain_succession(trace, activity1, activity2):
    if not act_in_trace(trace, [activity1]):
        return -1
    new_trace = []
    for i in range(len(trace)):
        new_trace.append(trace[i])
        if trace[i][event_id_tag] == activity1 and trace[i][
           lifecycle_tag] == "complete":
            new_event = {
                event_id_tag: activity2,
                lifecycle_tag: "complete",
                timestamp_tag: trace[i][timestamp_tag]
            }
            new_trace.append(new_event)
            break
    if i < len(trace) - 1:
        new_trace.extend(trace[i + 1:])
    trace[:] = new_trace
```

We also note that in our model we have non overlapping constraints, doing so, we avoid that the violation of a constraint cancels a different already applied one. For a more specific presentation refer to our repository LifeCycleAligner (2024).

5.2 Experiments

During our experiments, we fed the tool with the tuple $\langle p, \mathcal{L}, \mathcal{M} \rangle$ where p is the planner name used to run the problems and finding an optimal solution to the *life-cycle trace alignment problem*, $\mathcal{L}$ is the event log and $\mathcal{M}$ is the declare model expressed as .xml file. As we can see from the following table, we performed experiments under different parameters.

Parameter	**Values**
Constraints	1, 3, 5, 7
Trace Lengths	10, 20, 30
Noise Levels	0%, 10%, 20%, 30%
Processed Traces	20
Planners	fast-downward, symba2

Table 5.1: Test Settings

For the alphabet we used 9 activities, an empirically tested good trade-off between expressive power and time constraints.

`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`	`i`

Table 5.2: 9 Alphabet Activities

In our experiments, we adopted models with 1,3,5,7 constraints. In the table below, we reported all the 7 constraints (e.g, for 1 constraint we took the first, for two the first two and so on and so forth).

Active Constraints (8 Activities)	**Description**
`Not Existence:c`	Not Existence on activity c
`Existence:a`	Existence of activity a
`Not Chain Succession:b_b`	Not Chain Succession between b and b
`Existence:e`	Existence of activity e
`Not Chain Succession:d_d`	Not Chain Succession between d and d
`Response:f_g`	Response relation between f and g
`Not Succession:h_i`	Not Succession between h and i

Table 5.3: Active Constraints for 9 Activities

The tested log is generated with respect to all the possible combinations with:

- The model with the settled number of constraints (e.,g 1,3,5,7).
- The alphabet activities already shown (9).
- The trace length parameter (e.g 10,20,30).
- The noise percentage (e.g, 0% corresponds to no noise).

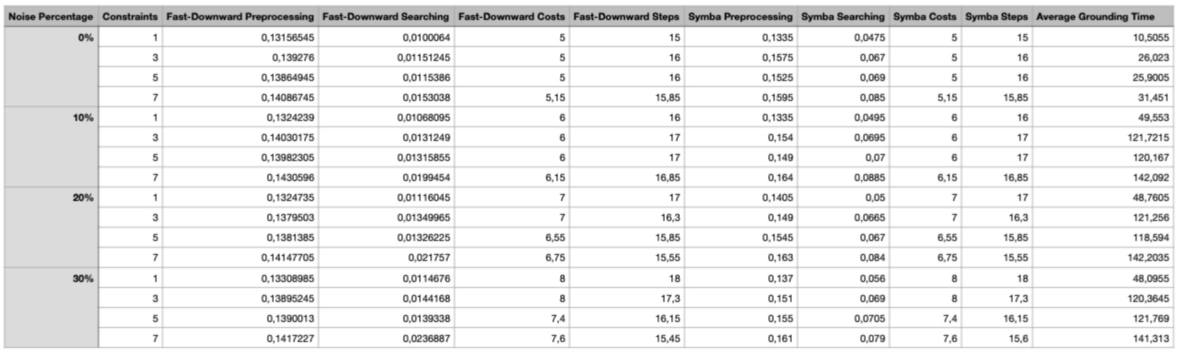
All the logs are initially generated with Rum tool Alman et al. (2020) with the specified alphabet and constraint model. Each log is formatted and noisified using are tool discussed in 5.1. Furthermore, we made the *LifeCycleAligner* tool process 20 traces per tested log. Obviously the noise is injected with respect to the current tested model, ensuring correct deviations are applied. For each test run combination, we used both planners. We tested on an Apple MacBook with an M2 chip, an 8-core processor, and 8 GB of RAM. For incompatibility issues (with symba2 planner) with the testing machine, we ran both planners with docker images hosted on docker containers that are dynamically created and shuttled-down avoiding waste of resources, saving the dump and result files. The tool also exposes the possibility to run the planners locally, this can be done by settling the `Container.local_planners` flag to *true*. The overall tests are performed using a bash-script that tests against all possible combinations of test parameters expressed before, that is $\langle p, \mathcal{L}_{\mathcal{M}_a,n,\eta(t,\mathcal{M}_a)}, \mathcal{M}_a \rangle$ where:

- p is the planner name.
- $\mathcal{L}_{\mathcal{M}_a,n,\eta(t,\mathcal{M}_a)}$ is the log with respect a model $\mathcal{M}_a$, the trace length n and the noise parameter.
- $\mathcal{M}_a$ the model with a the alphabet.

```
STARTING TESTS

----- RUNNING WITH: -----
Planner: fast-downward
Trace Length: 10
Noise: 20%
Trace Alphabet: 9
Constraints: 7
-------------------------

[###############################                ] 10/96
TIME ELAPSED: 15m 30s

EXIT_CODE: 0

...

----- RUNNING WITH: -----
Planner: symba2
Trace Length: 20
Noise: 10%
Trace Alphabet: 9
Constraints: 5
-------------------------

[##############################                 ] 20/96
TIME ELAPSED: 30m 15s

...

EXIT_CODE: 0

...

[##################################################] 96/96
ALL TESTS COMPLETED
```

5.3 Evaluation of the Results

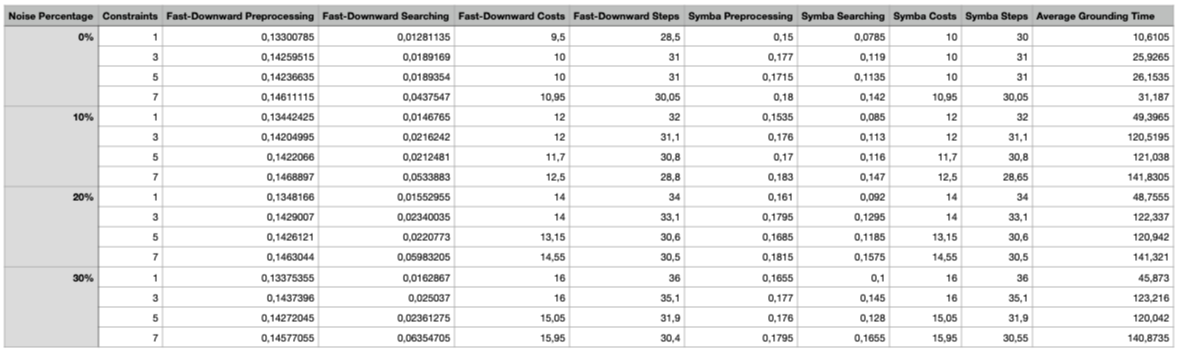
Before providing the results obtained in the previously stated test settings, we mention the evaluation metrics used to evaluate the outcomes. The **time** in seconds (s). We show different testing times, in particular, we concentrate on *LifeCycleAligner* grounding time, the time employed by the grounding algorithm to generate the .pddl files, the average pre-processing time, needed by the planner p to pre-process all the .pddl domain and problem files tuples for a particular test setting $\langle p, \mathcal{L}_{\mathcal{M}_a,n,\eta(t,\mathcal{M}_a)}, \mathcal{M}_a \rangle$ and the average (like for the pre-processing time) searching time, needed by the planner p to find the optimal solution to the problem. In addition, we measure the average (defined as before) of the planning steps required by the planner p to find the solution and the related cost of the *life-cycle alignment*. To parse the results, we created another ad-hoc script that parses each file and extract the required quantitative information, the reader can consult it at our usual repository LifeCycleAligner (2024). We tested with an alphabet size of 9 different activities, this means that up to different general activities a can be found in a trace t. We present the overall results below, noting that a pivot table was used for improved visualization.



Noise Percentage	Constraints	Fast-Downward Preprocessing	Fast-Downward Searching	Fast-Downward Costs	Fast-Downward Steps	Symba Preprocessing	Symba Searching	Symba Costs	Symba Steps	Average Grounding Time
0%	1	0,13156545	0,0100064	5	15	0,1335	0,0475	5	15	10,5055
	3	0,139276	0,01151245	5	16	0,1575	0,067	5	16	26,023
	5	0,13864945	0,0115386	5	16	0,1525	0,069	5	16	25,9005
	7	0,14086745	0,0153038	5,15	15,85	0,1595	0,085	5,15	15,85	31,451
10%	1	0,1324239	0,01068095	6	16	0,1335	0,0495	6	16	49,553
	3	0,14030175	0,0131249	6	17	0,154	0,0695	6	17	121,7215
	5	0,13982305	0,01315855	6	17	0,149	0,07	6	17	120,167
	7	0,1430596	0,0199454	6,15	16,85	0,164	0,0885	6,15	16,85	142,092
20%	1	0,1324735	0,01116045	7	17	0,1405	0,05	7	17	48,7605
	3	0,1379503	0,01349965	7	16,3	0,149	0,0665	7	16,3	121,256
	5	0,1381385	0,01326225	6,55	15,85	0,1545	0,067	6,55	15,85	118,594
	7	0,14147705	0,021757	6,75	15,55	0,163	0,084	6,75	15,55	142,2035
30%	1	0,13308985	0,0114676	8	18	0,137	0,056	8	18	48,0955
	3	0,13895245	0,0144168	8	17,3	0,151	0,069	8	17,3	120,3645
	5	0,1390013	0,0139338	7,4	16,15	0,155	0,0705	7,4	16,15	121,769
	7	0,1417227	0,0236887	7,6	15,45	0,161	0,079	7,6	15,6	141,313

Figure 5.2: Results Table for Trace Length 10.

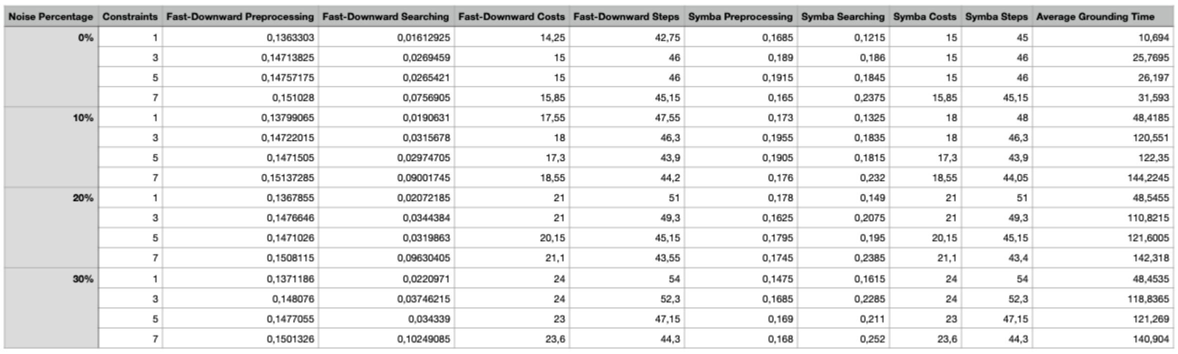
We have separated the overall table of the results for visualization purposes. We decided to segment the results based on the trace length parameter.



Noise Percentage	Constraints	Fast-Downward Preprocessing	Fast-Downward Searching	Fast-Downward Costs	Fast-Downward Steps	Symba Preprocessing	Symba Searching	Symba Costs	Symba Steps	Average Grounding Time
0%	1	0,13300785	0,01281135	9,5	28,5	0,15	0,0785	10	30	10,6105
	3	0,14259515	0,0189169	10	31	0,177	0,119	10	31	25,9265
	5	0,14236635	0,0189354	10	31	0,1715	0,1135	10	31	26,1535
	7	0,14611115	0,0437547	10,95	30,05	0,18	0,142	10,95	30,05	31,187
10%	1	0,13442425	0,0146765	12	32	0,1535	0,085	12	32	49,3965
	3	0,14204995	0,0216242	12	31,1	0,176	0,113	12	31,1	120,5195
	5	0,1422066	0,0212481	11,7	30,8	0,17	0,116	11,7	30,8	121,038
	7	0,1468897	0,0533883	12,5	28,8	0,183	0,147	12,5	28,65	141,8305
20%	1	0,1348166	0,01552955	14	34	0,161	0,092	14	34	48,7555
	3	0,1429007	0,02340035	14	33,1	0,1795	0,1295	14	33,1	122,337
	5	0,1426121	0,0220773	13,15	30,6	0,1685	0,1185	13,15	30,6	120,942
	7	0,1463044	0,05983205	14,55	30,5	0,1815	0,1575	14,55	30,5	141,321
30%	1	0,13375355	0,0162867	16	36	0,1655	0,1	16	36	45,873
	3	0,1437396	0,025037	16	35,1	0,177	0,145	16	35,1	123,216
	5	0,14272045	0,02361275	15,05	31,9	0,176	0,128	15,05	31,9	120,042
	7	0,14577055	0,06354705	15,95	30,4	0,1795	0,1655	15,95	30,55	140,8735

Figure 5.3: Results Table for Trace Length 20.

We also note that in the tables illustrated, the *Symba* column refers to the *Symba2** planner version and for the *Fast Downward* planner we used the latest version. The reader can consult the *Symba2** at[1] and at the original *Fast Downward* website[2].



Noise Percentage	Constraints	Fast-Downward Preprocessing	Fast-Downward Searching	Fast-Downward Costs	Fast-Downward Steps	Symba Preprocessing	Symba Searching	Symba Costs	Symba Steps	Average Grounding Time
0%	1	0,1363303	0,01612925	14,25	42,75	0,1685	0,1215	15	45	10,694
	3	0,14713825	0,0269459	15	46	0,189	0,186	15	46	25,7695
	5	0,14757175	0,0265421	15	46	0,1915	0,1845	15	46	26,197
	7	0,151028	0,0756905	15,85	45,15	0,165	0,2375	15,85	45,15	31,593
10%	1	0,13799065	0,0190631	17,55	47,55	0,173	0,1325	18	48	48,4185
	3	0,14722015	0,0315678	18	46,3	0,1955	0,1835	18	46,3	120,551
	5	0,1471505	0,02974705	17,3	43,9	0,1905	0,1815	17,3	43,9	122,35
	7	0,15137285	0,09001745	18,55	44,2	0,176	0,232	18,55	44,05	144,2245
20%	1	0,1367855	0,02072185	21	51	0,178	0,149	21	51	48,5455
	3	0,1476646	0,0344384	21	49,3	0,1625	0,2075	21	49,3	110,8215
	5	0,1471026	0,0319863	20,15	45,15	0,1795	0,195	20,15	45,15	121,6005
	7	0,1508115	0,09630405	21,1	43,55	0,1745	0,2385	21,1	43,4	142,318
30%	1	0,1371186	0,0220971	24	54	0,1475	0,1615	24	54	48,4535
	3	0,148076	0,03746215	24	52,3	0,1685	0,2285	24	52,3	118,8365
	5	0,1477055	0,034339	23	47,15	0,169	0,211	23	47,15	121,269
	7	0,1501326	0,10249085	23,6	44,3	0,168	0,252	23,6	44,3	140,904

Figure 5.4: Results Table for Trace Length 30.

From the above presented tables, we can see that the pre-processing time of both planners is consistently smaller than the time employed by the tool to ground the formulas. In other words, the tool helps the planner, feeding it with an already hardened version of the same planning problem. Thanks to this pre-computation, the search times of both planners benefits from it.

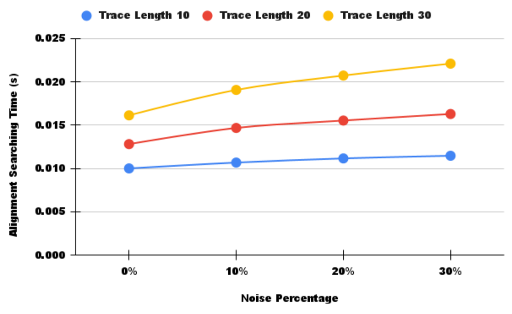


Figure 5.5: 1 Constraint Model Alignment Time.

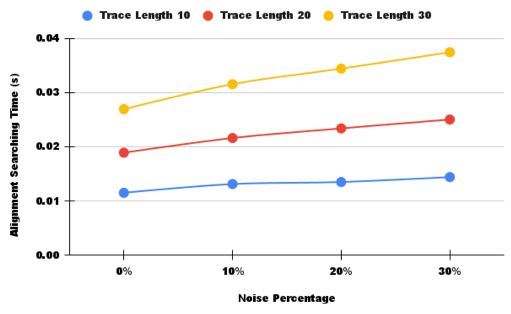


Figure 5.6: 3 Constraints Model Alignment Time.

We start by presenting the first figure 5.5, which shows the Alignment Search time in seconds relative to the Noise Percentage. As seen in 5.6, the searching time increases as the noise percentage rises, as expected. However, despite this trend, the alignment time remains very short, consistently staying under 0.04 seconds for each trace length and noise percentage in the models with 1 and 3 constraints.

[1] `https://fai.cs.uni-saarland.de/torralba/software.html`
[2] `https://www.fast-downward.org`

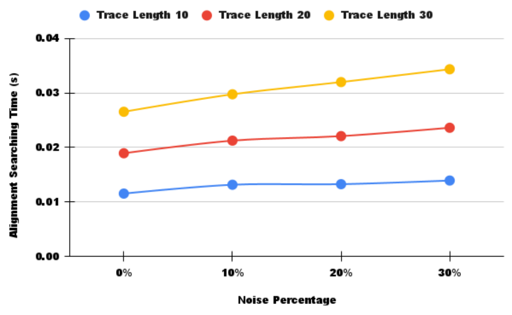


Figure 5.7: 5 Constraints Model Alignment Time.

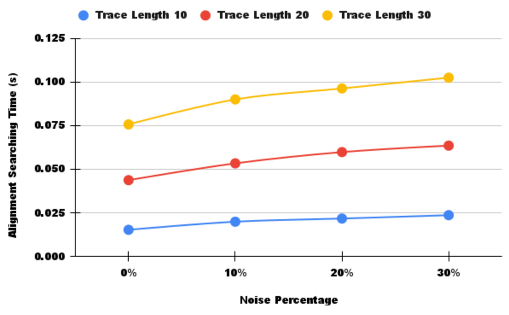


Figure 5.8: 7 Constraints Model Alignment Time.

This trend remains consistent in the plots shown in 5.7 and 5.8, where we observe a very low upper bound of 0.125 seconds. It is important to note that the results presented in these plots are for the *Fast-Downward* planner. This behavior also holds true for the *Symba2** planner, as we can see from the table 5.4.

Constraints	Grounding	Symba2*	Fast-Downward
1	38.97	0.09	0.01
3	96.44	0.13	0.02
5	97.16	0.13	0.02
7	114.27	0.16	0.06

Table 5.4: Average Time (s) for Grounding, Fast-Downward Searching, and Symba Searching for each Constraints Model.

The table above highlights the order of magnitude difference between the grounding time of the tool and the search time of each planner across the different constraint model variants (1, 3, 5, and 7). Another time, the searching time needed is non-significant with respect to the grounding time. This indicates that we have reached the purpose for which the tool has been created, that is decrementing significantly the number of states in the search space. Thanks to our tool, both planner accomplish the *life-cycle trace alignment* in less than one 1 second. Additionally, it emphasizes the similarity in results between the *Symba2** and *Fast-Downward* planners, with *Symba2** planner employing slightly more time. We can also verify that an increase in the constraints, leads to an increase in the grounding time as we expected, because of the state space growth. The increase is significant but the total grounding time is still acceptable.

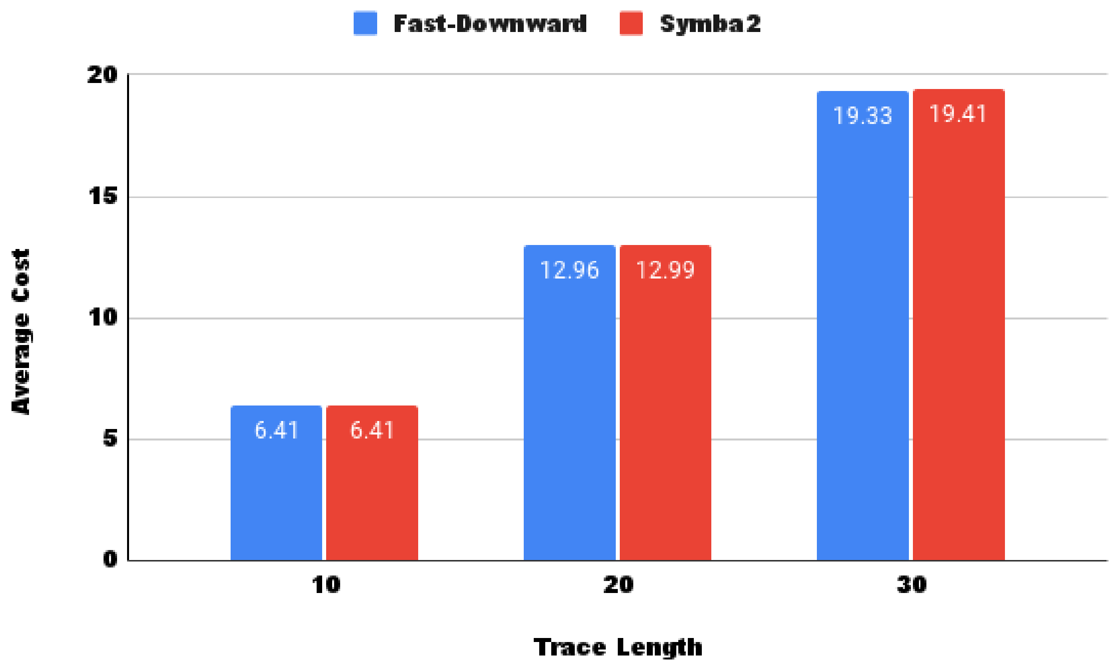


Figure 5.9: Average Cost for both planners in relation to Trace Length.

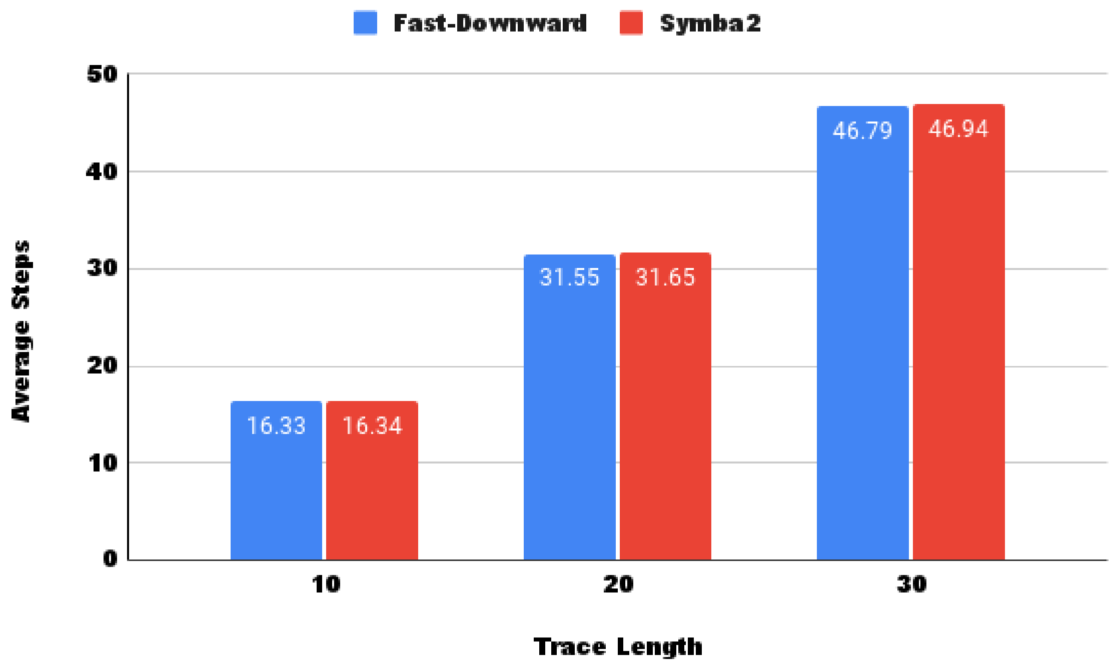


Figure 5.10: Average Steps for both planners in relation to Trace Length.

From the pictures 5.9 and 5.10, we can see that increasing the trace length leads to an increase in the average cost of both planners, this because the noise percentage, defined in 5.1 is always in relation to the trace length. This to say that, if we have non-zero noise, in a longer trace, we will have more deviations relating to a shorter one. This means that the planner will have to employ more *add* and *del* moves to align the trace.

To summarize, the two planners have similar performances on the different test settings, with *symba2* employing slightly more time in the case where the trace length is greater than 10. We can also see from the above pictures that the more the noise percentage increases in the event log, the more the searching time augments, as we could imagine before because of more alignment moves required to align the trace. We can make the same considerations for the trace length where a longer trace needs more alignments because the activity numbers grows and so the lifecycle corrections, but also the models corrections (in the case of non pure event logs where the noise percentage is greater than 0%). Moreover, as the model's constraints increase along with the noise, the search performances decline, though the degradation is minimal. It consistently remains significantly lower than the grounding and pre-processing time. In conclusion, we provide insights into the performance variations observed across different life-cycle models. In the *LifeCycleAligner* tool, the `combine` method is less memory-intensive as it avoids creating lifecycle DOT files. However, it makes the tool execution slower. Conversely, the DOT function variant requires more memory but is more efficient in terms of execution time for the tool. For the two default lifecycle models, the performance differences stem from their distinct automata structures. For instance, the planner requires less time to search with the chain-succession variant, but the tool takes more time to ground the problem. On the other hand, the alternate-succession variant reduces grounding time for the tool but increases the planner's search time. This is because the alternate-succession model is a more general lifecycle model, which can be satisfied in a greater number of ways and it also depends on the combination of relevant transitions that are created and that are related based on their labels. To summarize, as the life-cycle becomes more general and expressive, searching performance tends to degrade. Conversely, a more specific life-cycle results in improved performance for the planner search. In all cases, performance is highly dependent on the life-cycle model.

Chapter 6

Related Works

In this chapter, we review the related works that have primarily influenced the development of the research presented in this thesis. The literature on automated planning primarily focuses on algorithms that generate optimal plans and adapt to dynamic environments. Trace alignment, on the other hand, refers to the process of comparing and synchronizing the planned sequence of actions with the real execution traces. We can leverage the strengths of both with automated planning providing the foundation for generating efficient strategies, while trace alignment ensures these strategies remain aligned with a predefined model. The chapter is organized into three main sections: in 6.1 we concentrate on Automated Planning for BP, in 6.2 we mention some research around Automated Planning for Trace Alignment with declarative process models and in 6.3 we center the discussion around implementation tools for Trace Alignment.

6.1 Automated Planning for BP

We have already discussed Business Process Management and its importance in the process life-cycle, particularly in terms of streamlining operations and ensuring efficiency across various stages of execution. The research presented in Marrella (2018) emphasizes the need for automated solutions in Business Process Management (BPM) systems. Specifically, it focuses on developing a methodology to translate concrete problems in the BPM domain into planning problems. By leveraging the capabilities of automated planning, these BPM challenges can be effectively tackled, enabling more automation and support to the BPM field. In the conformance checking context, Bergami et al. (2021) explores the challenges of aligning event logs in relation to declarative process models. Their work demonstrates how the trace alignment problem can be reduced to a data-agnostic problem, ensuring correct solutions. Additionally, they propose leveraging automated planning techniques to align traces with data-aware models, thus enabling automated alignment strategies for complex business processes. In Bernardi et al. (2016) the authors highlight the problem of having non-atomic activities within a process, they put this further showing how existing techniques for discovering declarative process models can be adapted to handle business processes that involve non-instantaneous activities. By leveraging the information available in the logs, they focus on identifying business rules whose validity is influenced by the characteristics of these life-cycles.

6.2 Automated Planning with Declarative Models

Our planning technique is mainly built upon the breakthrough study in De Giacomo et al. (2017). It presents how to represent a declarative model formalized as LTLf 2.3.1 and a trace of an event log as two automata: the constraint automaton and the trace automaton. Next, it introduces automata augmentations to make them suitable for trace alignment, adding *del* and *add* transitions for each activity in the trace. They also provide the encoding in Planning Domain Definition Language (PDDL) of the planning problem that allows the synthesis of optimal alignment plans in an automated way, providing a significant contribution to this field. In Giacomo et al. (2020b) this concepts is extended to DECLARE models, finding also in this case a suitable encoding in 2.4.4. Another research that very closely relates to the arguments of this research is Acitelli et al. (2020) where the traditional trace alignment is extended to construct optimal alignments guided by specialized cost models, designed as automata, that assign context-sensitive variable costs to deviations.

6.3 Trace Alignment Tools

In this section of the related research, we present some tool implementations. In Giacomo et al. (2020a) the authors implemented a tool for Trace Alignment. The *TraceAligner* is a Java tool for declarative models that takes a business process (BP) log in XML or XES format and a set of LTLf/LDLf constraints as input and generates cost-optimal planning instances in PDDL, where each instance represents a log trace, and its solution corresponds to an optimal alignment of the trace with respect to the specifications. Once the problem instances are created, they tested it with different state of the art cost optimal planners. In de Leoni and Marrella (2017) the authors designed and implemented a planning-based alignment tool for imperative processes. It provides a GUI for creating or importing event logs and Petri nets, customizing cost functions and markings and selecting search heuristics. It supports XES and

PNML formats for event logs and Petri nets. Moreover, it generates a PDDL domain file and problem files for each event log trace, which are solved by a PDDL planner to achieve optimal alignment.

Chapter 7

Conclusions and Future Remarks

Business Process Management (BPM) is a discipline that focuses on analyzing, designing, optimizing, and managing business processes to improve organizational efficiency and effectiveness. *Business* efficiency involves maximizing the use of time, effort, and resources across functional departments. By streamlining operations, organizations can achieve comparable or improved results with fewer resources, reducing costs and enhancing the return on investment in their operations. *Business* effectiveness refers to the quality of outcomes achieved from the investment of resources in functional departments. Procedural and static process models may be inadequate to describe the expected behaviors that those processes should follow because of its intrinsic dynamical aspects that we have previously discussed in 2.1 and 2.3.

In particular, DECLARE provides a series of templates designed to formalize organizational constraints. Conformance checking addresses the challenge of determining whether process executions adhere to the rules established in the process model. We have two sides of the same coin: a *process model*, which defines the rules, and an *event log*, which captures actual process executions. Conformance checking techniques assess whether the observed executions align with the process model.

A robust framework for conformance checking is Trace Alignment. It allows for identifying specific deviations responsible for nonconformity by aligning traces from the event log with those in the model. This alignment highlights discrepancies, making deviations clearly identifiable. In the existing works, DECLARE is used to describe *inter-activities-life-cycles* within the same process instance. In this thesis we progressed further analyzing not only inter-activities-life-cycles but also *intra-activities-life-cycles*. This because in many processes activities exhibit a non atomic and instantaneous behavior and cannot be treated as a whole. In this work, we have addressed the problem of *aligning the life-cycles of activities* with respect to a life-cycle model formalized as a Deterministic Finite Automaton (DFA). We embed activity-life-cycle corrections in addition to the standard alignment, which can be interpreted as aligning the process activities with a process model that serves as the life-cycle model of the process itself. We list the key contributions of this thesis:

- The *definition* of the Life-Cycle Trace Alignment, as an extension of the traditional approach. We clearly re-define what an activity is in our context, because now a *general activity* has a extension in a *life-cycle-activity.* The concept of the *Life-Cycle* is embedded into a Deterministic Finite Automaton (DFA) and it represents *intra-activity* constraints.
- The *formalization* of this Life-Cycle Trace Alignment as a planning problem. We have constraints-DFA side by side to life-cycle-DFA as the model part of our problem (in addition to the trace-DFA). Next we have augmented these automata like we discussed in 2.
- The PDDL Encoding of this problem. We have extended the traditional encoding to also handle *activity Life-Cycle.*
- The *LifeCycleAligner Tool* is implemented. The tool can generate problems that tracks also the activities life-cycles.

In this work, we build upon the theoretical definition of the activity life-cycle trace alignment problem by extending the traditional atomic view of activities to incorporate the ability to address *intra-activity* relationships. This approach proves beneficial in real-world settings where activities frequently consist of multiple life-cycle states. By addressing intra-activity relationships, the proposed approach allows for a more fine-grained understanding and alignment of activities, which can be particularly advantageous in multiple scenarios. This flexibility enhances the ability to model real-world processes, where interactions within the same activity often play a critical role in determining outcomes. We effectively inspect the activity information that we find in the Event Log to provide a more detailed aligning that is based on a pre-defined model. Highlighting these deviations offers a more precise insight into where and when misalignments occur, enabling the organization to adjust and restructure its processes effectively to address them. Using the information about the activity life-cycle we can effectively accomplish this variant of the trace alignment task and gaining more profound insights about issues within the scope of a particular activity. In 3, we have formulated the life-cycle problem as a planning problem that allowed us to perform the PDDL Encoding. In 4 we have provided the implementation of our theoretical derivations. The tool provides a CLI that is straight-forward to use and that can take in input different constraints model formats to efficiently manage the creation and execution of the PDDL-encoded problems. Furthermore, the tool allows to run the created problems into two different state-of-the-art planners that one can choose depending on their needs. All the grounding effort is placed on the tool so that any of the tested planners can perform the search efficiently. Moreover in 5 we performed the evaluation on the proposed tool. We have tested under different conditions and the results have shown that, despite mid-to-high grounding times, the pre-processing and searching times for every planner is negligible.

7.1 Future Remarks

Finally we addressed several opportunities for eventual upcoming research. Future investigations could focus on extending this *intra-activity* interest. Other researches could leverage different activity-related fields present in the Event Log like the resource or the timestamp fields to ensure other types of constraints enforcement. The cost of the activity life-cycle correction could be dependent on the life-cycle state in which we are or on other intra activity life-cycle relations. The tool can be extended to speed up the grounding time phase by leveraging some additional data-structures. More planner can be linked to the tool by only modifying the *Runner* class so that more state-of-the art can be used to produce the activity life-cycle trace alignment. The tool can accept more formats for the constraint model part. This could be done by only modifying the *Loader* class providing it additional loading methods for the different formats required. Moreover, future research could focus on extending the life-cycle model to support parallel activity life-cycle correction, where multiple life-cycles of the same activity can interleave between each other, providing additional flexibility and level of detail to model real-world situations. More specific solutions could define activity-specific custom life-cycles, augmenting the complexity of the implementation and its expressive power. Advanced evaluation can be performed on different types of life-cycle models, overcoming the one composed by *assigned,started* and *completed* discussed in this work.

Bibliography

Giacomo Acitelli, Marco Angelini, Silvia Bonomi, Fabrizio M. Maggi, Andrea Marrella, and Alessandro Palma. Context-aware trace alignment with automated planning. *Sapienza Universita di Roma, Rome, Italy, Free University of Bozen-Bolzano, Bozen, Italy*, 2020. Email: {acitelli, angelini, bonomi, marrella, palma}@diag.uniroma1.it, maggi@inf.unibz.it.

Anti Alman, Claudio Di Ciccio, Dominik Haas, Fabrizio Maria Maggi, and Alexander Nolte. Rule mining with RuM. In *ICPM*, pages 121–128. IEEE, 2020.

Giacomo Bergami, Fabrizio Maria Maggi, Andrea Marrella, and Marco Montali. Aligning data-aware declarative process models and event logs. In *Business Process Management: 19th International Conference, BPM 2021, Rome, Italy, September 06–10, 2021, Proceedings 19*, pages 235–251. Springer, 2021.

BPM Offensive Berlin. Bpmn 2.0 business process model and notation, poster, 2011.

Mario Bernardi, Marta Cimitile, Chiara Di Francescomarino, and Fabrizio Maggi. Do activity lifecycles affect the validity of a business rule in a business process? *Information Systems*, 62, 06 2016. doi: 10.1016/j.is.2016.06.002.

Joos Buijs. Receipt phase of an environmental permit application process ('wabo'), coselog project, 2014. URL `https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6`. Accessed: 2024-09-19.

Jeanine Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 7, pages 215–249. ACM, 1998.

Giuseppe De Giacomo, Moshe Y Vardi, et al. Linear temporal logic and linear dynamic logic on finite traces. In *Ijcai*, volume 13, pages 854–860, 2013.

Giuseppe De Giacomo, Moshe Y Vardi, et al. Synthesis for ltl and ldl on finite traces. In

Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, pages 1558–1564. AAAI Press, 2015.

Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. On the Disruptive Effectiveness of Automated Planning for LTLf-Based Trace Alignment. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3555–3561, 2017. URL `https://hdl.handle.net/11573/965532`.

M. de Leoni and A. Marrella. Aligning real process executions and prescriptive process models through automated planning. *Eindhoven University of Technology, The Netherlands, Sapienza University of Rome, Italy*, 2017. Received 17 September 2016, Revised 21 March 2017, Accepted 22 March 2017, Available online 31 March 2017.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1): 269–271, December 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL `http://dx.doi.org/10.1007/BF01386390`.

Giuseppe De Giacomo, Francesco Fuggitti, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. A tool for declarative trace alignment via automated planning. *University of Oxford, Oxford, UK, Sapienza University, Rome, Italy, York University, Toronto, Canada, Free University of Bozen-Bolzano, Bolzano, Italy*, 2020a.

Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Sebastian Sardina. Computing trace alignment against declarative process models through planning. *Sapienza - Universita di Roma, Italy, University of Tartu, Estonia, RMIT University, Melbourne, Australia*, 2020b. degiacomo@dis.uniroma1.it, f.m.maggi@ut.ee, marrella@dis.uniroma1.it, sebastian.sardina@rmit.edu.au.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, volume 4. IEEE, 1968.

M. Helmert. The Fast Downward Planning System. *J. Artif. Int. Res. (JAIR)*, 26(1):191–246, 2006. doi: 10.1613/jair.1705.

IEEE Task Force on Process Mining. Xes standard - extensible event stream. `http://www.xes-standard.org`, 2023. Accessed: 2024-09-19.

LifeCycleAligner. LifeCycleAligner, 2024. URL `https://github.com/GianmarcoBordin/LifeCycleAligner`.

Andrea Marrella. Automated planning for business process management. *Springer-Verlag GmbH*, 2018. Received: 2 March 2018 / Revised: 24 July 2018 / Accepted: 21 October 2018.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl—the planning domain definition language. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*. AAAI Press, 1998.

Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77 (4):541–580, 1989.

M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *EDOC*, pages 287–300, 2007.

Maja Pesic and Wil M. P. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws, Vienna, Austria, September 4-7, 2006, Proceedings*, pages 169–180, 2006. doi: 10.1007/11837862_18. URL `https://doi.org/10.1007/11837862_18`.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 46–57. IEEE Computer Society, 1977.

Alvaro Torralba, Vidal Alcazar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp. Symba: A symbolic bidirectional a planner. In *International Planning Competition*, pages 105–108, 2014.

W. van der Aalst, M. Pesic, and H. Schonenberg. Declarative Workflows: Balancing Between Flexibility and Support. *Computer Science - R&D*, 23(2):99–113, 2009.

Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Publishing Company, Incorporated, 2nd edition, 2012. ISBN 9783642286162.