Beyond the Frontiers of Timeline-based Planning

De Benedictis, Riccardo

http://hdl.handle.net/10026.1/14236

http://dx.doi.org/10.24382/798

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.
BEYOND THE FRONTIERS OF TIMELINE-BASED PLANNING

by

RICCARDO DE BENEDICTIS

A thesis submitted to University of Plymouth in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing, Electronics and Mathematics

March 2019
Copyright Statement

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author’s prior consent.
Acknowledgments

I would like to express my deep gratitude to Dr. Amedeo Cesta and Prof. Angelo Cangelosi, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this research work. I would also like to thank Dr. Gabriella Cortellessa, for her extremely valuable and constructive suggestions during the planning and development of this research work.

A special thank to the Institute of Cognitive Sciences and Technologies of CNR for the stimulating multidisciplinary environment and for the rooms made available. I am pleased to extend greetings to all the members of the Planning and Scheduling Technology Laboratory (PST) at ISTC-CNR for their useful feedbacks given every time needed.

I would like to immensely thank Hector Geffner, along with his outstanding team at the Pompeu Fabra University in Barcelona, for the opportunity he gave me. A special thank goes also to Dimitri Ognibene and to Eloisa Vargiu for their pleasant company and for the interesting discussions we had (and, nevertheless, for physically allowing my early livelihood in Barcelona :).

Last but not least, I am very grateful to my family, for withstanding the piles of papers, to the members of the Trio Bottega, for the beautiful played notes, to the crew, for the evenings spent together, and to the wonderful people who, sometimes, you can meet on the mountains.
Author’s Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Doctoral College Quality Sub-Committee.

Work submitted for this research degree at the University of Plymouth has not formed part of any other degree either at the University of Plymouth or at another establishment.

Relevant seminars and conferences were regularly attended at which work was often presented; related papers have been prepared for publication and published. I have been member of conferences and workshops program committees. I have worked in several European projects.

The following external institutions were visited for consultation purposes:

– Universitat Pompeu Fabra (UPF)

Core Publications

Journals

DOI: https://doi.org/10.3233/IA-160100

Book Chapters

DOI: https://doi.org/10.1007/978-3-642-36907-0_15

Presentations at conferences

International Conferences with Official Proceedings

DOI: https://doi.org/10.1007/978-3-319-24309-2_32

DOI: https://doi.org/10.5220/0003746901440153
Workshops and Symposia Proceedings


DOI: https://doi.org/10.1007/978-3-319-18374-9_21

DOI: https://doi.org/10.3233/978-1-61499-411-4-16

Related Publications

The following publications describe those works which have been carried out by Riccardo during the Ph.D. period that, in a more or less significant way, contributed to the development of the thesis.

Part of these publications (i.e., [J1, CO3, CO2]) are related to the PANDORA project, a European project (GA n. 225387) which led to the creation of an intelligent simulation environment for personalized training of managers in crisis situations. A virtual room was used for the simulation of realistic situations in a context of crisis in order to allow the planning of the training of managers taking into account the different user profiles and emotional factors such as stress, anxiety, etc. In this project, Riccardo has developed the underlying reasoners, described in a broader way in this thesis, constituting two of the three main modules of the project.

Another part of these publications (i.e., [J3, CP2, CO7, CO5, WS9, WS8, WS5, WS4, WS2]) are related to the GiraffPlus project, a European Project (GA n. 288173) which led to the creation of a complex system allowing monitoring activities of elderly people in their homes through a network of sensors placed in the same house, around the house and also on the body of the elder. The sensors can measure the blood pressure and other physiological parameters or identify falls, gas leaks, etc. Different services, depending on the specific needs of the user, can be tailored to their specifications defined by both the elder and the health care professional who is responsible for monitoring the status of the elderly. In this project, Riccardo has been responsible for the development of the user modeling, aiming at the personalization services, and for
the planning aspects required for the reminding service. Such services are based on the topics described in this thesis.

The reminding of these publications describe some Space related activities (i.e., [CO4, WS3, WS1]) strongly relying on timeline-based reasoning, the personalized assignment of activities to volunteers (i.e., [CO8, WS12]) relying on the module described in Section 4, and the development of a timeline-based Intelligent Tutoring System (i.e., [WS13]). Each of these activities exploit different parts of the results described in this thesis.

**Journals**

DOI: [https://doi.org/10.1089/tmj.2016.0258](https://doi.org/10.1089/tmj.2016.0258)

DOI: [https://doi.org/10.1016/j.knosys.2013.11.011](https://doi.org/10.1016/j.knosys.2013.11.011)

**Book Chapters**

DOI: [https://doi.org/10.1007/978-3-319-54283-6_25](https://doi.org/10.1007/978-3-319-54283-6_25)

**Presentations at conferences**

**International Conferences with Official Proceedings**

DOI: [https://doi.org/10.1007/978-3-319-70169-1_34](https://doi.org/10.1007/978-3-319-70169-1_34)

DOI: [https://doi.org/10.1109/CloudNet.2016.15](https://doi.org/10.1109/CloudNet.2016.15)
DOI: https://doi.org/10.1007/978-3-319-22689-7_45

DOI: https://doi.org/10.1007/978-3-642-38577-3_20

URL: https://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6038

DOI: https://doi.org/10.3233/978-1-61499-098-7-1001

Workshops and Symposia Proceedings


DOI: https://doi.org/10.1007/978-3-319-18374-9_3


Word count of main body of thesis: 73,940
Abstract

Any agent, either biological or artificial, understands how to behave in its environment according to its prior knowledge and to its prior experience. The process of deciding which actions to undertake and how to perform them so as to achieve some desired objective is called deliberation. In particular, planning is an abstract and explicit deliberation process that chooses and organizes actions, by anticipating their expected outcomes, with the aim to achieve, as best as possible, some pre-stated objectives called goals. Among the most widespread approaches to automated planning, the classical approach broadly pursues the following definition of planning: starting from a description of the initial state of the world, a description of the desired goals, and a description of a set of possible actions, the planning problem consists in synthesizing a plan, i.e., a sequence of actions, that is guaranteed, when applied to the initial state, to generate a state, called a goal state, which contains the desired goals.

In order to cope with computational complexity, however, the classical approach to planning introduces some restrictive assumptions. Among them, for example, there is no explicit model of time and concurrency is treated only roughly. Additionally, goals are specified as a set of goal states, therefore, objectives such as states to be avoided and constraints on state trajectories or utility functions are not handled. In order to relax these restrictions, some alternative approaches have been proposed over the years. The timeline-based approach to planning, in particular, represents an effective alternative to classical planning for complex domains requiring the use of both temporal reasoning and scheduling features. This thesis focuses on timeline-based planning, aiming at solving some efficiency issues which inevitably raise as a consequence of the drop out of these restrictions. Regardless of the followed approach, indeed, it turns out that automated planning is a rather complex task from a computational point of view. Furthermore, not all of the approaches proposed in literature can rely on effective heuristics for efficiently tackling the search. This is particularly true in the case of the more recent and hence less investigated timeline-based formulation. Most of the timeline-based planners, in particular, have usually neglected the advantages triggered in classical planning from the use of Graphplan and/or modern heuristic search, namely the capability of reasoning on the whole domain model. This thesis aims at reducing the performance gap between the classical approach at planning and the timeline-based one. Specifically, the overall goal is to improve the efficiency of timeline-based reasoners taking inspiration from techniques applied in more classical approaches to planning. The main contributions of this thesis, therefore, are a) a new formalism for timeline-based planning which overcomes some limitations of the existing ones; b) a set of heuristics, inspired by the classical approach, that improve the performance of the timeline-based approach to planning; c) the introduction of sophisticated techniques like the non-chronological backtracking and the no-good learning, commonly used in other fields such as Constraint Processing, into the search process;
d) the reorganization of the existing solver architectures, of a new solver called ORATIO, that allows to push the reasoning process beyond the sole automated planning, winking at emerging fields like, for example, Explainable AI and e) the introduction of a new language for expressing timeline-based planning problems called RIDDLE.
# Contents

1 Introduction ........................................ 1
   1.1 Different approaches at automated planning ........ 3

2 An Introduction to Automated Planning and some of its Challenges 5
   2.1 Classical planning .................................. 8
      2.1.1 Heuristics for classical planning .............. 15
   2.2 Partial-order planning .............................. 18
      2.2.1 An algorithm for partial-order planning ....... 22
   2.3 Planning with timelines ............................ 24
      2.3.1 Interactions among tokens: the timelines ...... 30
      2.3.2 Algorithms for timeline-based planning ...... 33
      2.3.3 Timeline-based planning vs other approaches .. 36
   2.4 The performance gap ............................... 39

3 Narrowing the Performance Gap with tLOC 41
   3.1 The integrated Logic and Constraint Reasoner (tLOC) 41
   3.2 Preliminary heuristics for tLOC .................... 45
      3.2.1 The ALLREACHABLE heuristic ................. 46
      3.2.2 The MINREACH heuristic ....................... 47
   3.3 Experimental evaluation ........................... 47
      3.3.1 Results ........................................ 50
   3.4 The gap does not narrow enough ................... 52

4 The SMT-based Constraint Network 55
   4.1 The SAT core ....................................... 57
      4.1.1 The solver state ................................ 58
      4.1.2 Variables and constraints ..................... 60
      4.1.3 Reified constraints ............................ 62
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4</td>
<td>Theories</td>
<td>63</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Propagation</td>
<td>63</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Learning</td>
<td>65</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Search</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>The Linear Real Arithmetic (LRA) theory</td>
<td>69</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Preprocessing</td>
<td>70</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The Application Programming Interface (API)</td>
<td>71</td>
</tr>
<tr>
<td>4.2.3</td>
<td>The solver state</td>
<td>72</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Variables and constraints</td>
<td>74</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Updating variables and pivoting the tableau</td>
<td>77</td>
</tr>
<tr>
<td>4.2.6</td>
<td>The \texttt{check()} procedure</td>
<td>78</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Theory propagation</td>
<td>80</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Backtracking</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>The Object Variables theory</td>
<td>84</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Variables and constraints</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>The constraint network</td>
<td>88</td>
</tr>
<tr>
<td>4.5</td>
<td>The Critical-Path Heuristics and the ORATIO Solver</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>The ORATIO architecture</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Building the causal graph</td>
<td>92</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Representing flaws</td>
<td>95</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Representing resolvers</td>
<td>97</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Building the graph</td>
<td>98</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Updating flaws’ and resolvers’ estimated costs</td>
<td>99</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Choosing the right flaw and the right resolver</td>
<td>100</td>
</tr>
<tr>
<td>5.2.6</td>
<td>The causal graph applied to the rover domain</td>
<td>101</td>
</tr>
<tr>
<td>5.3</td>
<td>The new solving algorithm</td>
<td>103</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The role of pruning</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>Increasing the heuristic accuracy</td>
<td>105</td>
</tr>
<tr>
<td>5.5</td>
<td>Different flaws (and their resolvers)</td>
<td>105</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Object variable and disjunction flaws</td>
<td>106</td>
</tr>
<tr>
<td>5.5.2</td>
<td>The atom flaw</td>
<td>106</td>
</tr>
<tr>
<td>5.6</td>
<td>Defining the timelines</td>
<td>109</td>
</tr>
<tr>
<td>5.6.1</td>
<td>The state-variable</td>
<td>109</td>
</tr>
<tr>
<td>5.6.2</td>
<td>The reusable resource</td>
<td>111</td>
</tr>
<tr>
<td>5.7</td>
<td>Experimental results</td>
<td>111</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Results</td>
<td>112</td>
</tr>
<tr>
<td>6.1</td>
<td>The RIDDLE Language</td>
<td>121</td>
</tr>
<tr>
<td>6.1.1</td>
<td>An Object-Oriented language</td>
<td>122</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Identifiers</td>
<td>122</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Primitive types</td>
<td>123</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Operators</td>
<td>125</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Complex types</td>
<td>127</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Type inheritance</td>
<td>129</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Existentially scoped variables</td>
<td>129</td>
</tr>
</tbody>
</table>
xiv
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An autonomous agent perceives the environment through sensors, it deliberates according to its prior knowledge and experience and acts upon that environment through actuators.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>A state transition system example.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A restricted state transition system example.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The classic rover domain.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>A non-deterministic forward search planning algorithm.</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>A non-deterministic backward search planning algorithm.</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>A partial plan for the classic rover domain.</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>The PSP procedure.</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>Different types of timelines: a symbolic timeline, a piecewise constant timeline and a continuously changing timeline.</td>
<td>26</td>
</tr>
<tr>
<td>2.9</td>
<td>The rover domain represented through timelines.</td>
<td>30</td>
</tr>
<tr>
<td>2.10</td>
<td>Different kinds of timelines.</td>
<td>31</td>
</tr>
<tr>
<td>2.11</td>
<td>The Timeline-based Planning (TP) procedure.</td>
<td>34</td>
</tr>
<tr>
<td>2.12</td>
<td>A partial plan for the rover domain.</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>A high-level view of the iLoc reasoning engine.</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>The static causal graph associated to the rules of the rover domain and the costs estimated by the ALLREACHABLE heuristic.</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>The AND/OR static causal graph associated to the rules of the rover domain and the costs estimated by the MINREACH heuristic.</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Blocks world.</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>Temporal machine shop.</td>
<td>51</td>
</tr>
<tr>
<td>3.6</td>
<td>Cooking carbonara.</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Implication graph containing a conflict.</td>
<td>66</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

4.2 Propagating a lower bound update of $x$ such that $lb(x) > c$ in an $x \leq c$ assertion. The propagation results in assigning to the variable $b$ the value FALSE, in case of an assertion, and the value TRUE, in case of an assertion. ........................................ 82

5.1 The topology of the causal graph is exploited for recognizing a critical path. ........................................ 90

5.2 The overall ORATIO architecture. ........................................ 91

5.3 UML class diagram of the new solver. ........................................ 93

5.4 The causal graph generated from the rover domain. .................. 102

5.5 An example of cyclic causal graph. ........................................ 109

5.6 Execution time of different ORATIO implementations on the tower domain. ........................................ 113

5.7 Blocks world solving statistics. Percentage of created flaws (a) in the inaccurate case and in (b) the accurate one. Percentage of solved flaws (c) in the inaccurate case and in (d) the accurate one. Percentage of execution time (e) in the inaccurate case and in (f) the accurate one. 114

5.8 Execution time of different solvers on the tower domain. ........... 115

5.9 Execution time of different solvers on the cooking carbonara domain. 116

5.10 Cooking carbonara solving statistics. (a) Percentage of created flaw. (b) Percentage of solved flaws. (c) Percentage of execution time. 116

5.11 Execution time of different solvers on the temporal machine shop domain. 117

5.12 Execution time of different solvers on the Goal Oriented Autonomous Controller (GOAC) domain. 118

6.1 Structuring the code in different compilation units. ................. 122

7.1 Scooter ignition troubleshooting. ........................................ 146
An agent (from the Latin agere, to do) is an autonomous entity which acts within a given environment. In particular, as depicted in Figure 1.1, an autonomous agent is anything that can, autonomously, perceive its environment through sensors and act upon that environment through actuators. A human agent, for example, has eyes, ears, and other organs as sensors for perceiving the environment, and hands, legs, mouth, and other body parts as actuators for acting upon it. In a similar way, a robotic agent might have cameras and infra-red range finders as sensors and several motors as actuators. Moreover, a software agent might receive text messages, sensory inputs or files and act on the environment by displaying content on a screen, by writing files or by sending network packets.

Intuitively, in order to act in an effective manner, any agent should understand how to behave in its own environment according to its prior knowledge and, possibly, its prior experience. The process of deciding which actions to undertake and how to perform them so as to achieve some desired objective is called deliberation. Specifically, deliberation consists in the reasoning process which, by exploiting the knowledge of the environment and the prior experience of the agent, allows to simulate what will happen if the agent performs some actions. In particular, through the study of Artificial Intelligence (see [96]) and, more specifically, of automated planning, an area of Artificial Intelligence that studies this deliberation process from a computational point of view [58], we are interested in investigating the deliberation capabilities that allow artificial agents to reason about their actions, how to choose them, how to organize them purposefully, and how to deliberately act so as to achieve some desired objectives. Planning, specifically, is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. This particular deliberation process aims at achieving, as best as possible, some pre-stated objectives called goals.

Depending on the environment the agent acts upon, deliberation must take into account different aspects. Sensors, for example, might not provide a complete and precise access to the environment. According to this aspect, indeed, the environment can be distinguished into fully observable and partially observable. Examples of fully ob-
servable environments are the board games, in which the state of the world is perfectly defined and known. A robotic agent which acts in a real environment, on the contrary, is likely to perceive it as partially observable, since sensors might be noisy and inaccurate or, simply, because parts of the state are missing from the sensor data. Furthermore, if the next state of the environment is completely determined by the current state and by the action taken by the agent, the environment is said deterministic. An example of deterministic environment is a software agent which tries to solve a fifteen puzzle. It might happen, however, that the environment is non deterministic either because it is partially observable or because the outcome of the agent’s actions might be uncertain. Another factor to be taken into account in the description of the environment is whether the environment is episodic or sequential. In episodic environments the agent’s experience is divided into atomic episodes. Each of these episodes is characterized by a perception and by a single action. The crucial aspect is that the next episode is not dependent from the previous action. An example of episodic environment is an agent which performs a classification action since, typically, classifying an object does not affect the classification of another object. In sequential environments, however, the outcome of an action might influence the future decisions. An example of sequential environment is an agent which plays chess. Finally, the last aspect to be taken into account is whether the environment is (perceived as) discrete or continuous. In the former case, the duration of the actions does not affect (or, in the case, it does negligibly) the behavior of the agent. Examples of discrete environments are, again, board games, in which the duration of moving a piece can be neglected. Taking time into account, however, might be important in some scenarios. A typical example in which time is important, indeed, is when preparing a meal, since it might avoid burning the food is being cooked.
1.1 Different approaches at automated planning

Since there are various types of actions, there are also various forms of planning. Examples include path and motion planning, perception planning and information gathering, navigation planning, manipulation planning, communication planning, and several other forms of social and economic planning. A natural approach for these diverse forms of planning is to address each problem with the specific representations and techniques adapted to the problem. Although these domain-specific approaches are certainly well justified and highly successful in most of their specific application areas, they present some weakness:

- Some commonalities among these forms of planning are not addressed in the domain-specific approaches. The study of these commonalities is needed for understanding the process of planning and it may help to improve the domain specific approaches.

- It is more costly to address each planning problem anew instead of relying on and adapting some general tools.

- Domain-specific approaches are not satisfactory for studying and designing an autonomous intelligent machine. Its deliberative capabilities will be limited to areas for which it has domain-specific planners, unless it can develop by itself new domain-specific approaches from its interactions with the environment.

The domain-independent approach to planning aims at mitigating the above weaknesses. For solving a particular problem, a domain independent planner takes as input the problem specification and the knowledge about its domain synthesizing a set of actions which, starting from some given situation, if executed, would achieve the desired goals. Compared to the domain-specific one, for which it is easier to exploit domain specific knowledge, the domain-independent approach, however, being more general, has to deal with issues related to performance. Without relying on carefully designed heuristics, indeed, such approaches can easily generate solutions in an unreasonable amount time, severely affecting their usefulness.

Additionally, while remaining within the domain-independent case, the different choices in representing the environment and the actions, as well as the adopted strategies for performing the reasoning process, have led, over the years, to the emergence of different approaches at automated planning. Based on mathematical logic, the “classic” approach to automated planning is the oldest one and, till now, the most common approach within the automated planning research community. Despite the interesting results achieved in this field over the years, classical planning presents some limitations which make it quite cumbersome to apply in most of the real-world applications. The environment, for example, as will be explained in greater detail later on, is typically considered as fully observable, deterministic, sequential and discrete. Actions, for example, are seen as atomic entities that are taken one at a time in a sequence. As a consequence, with the years, several extensions to classical planning have been proposed aimed at facilitating its application in real contexts and, in parallel, new approaches
have emerged adopting a different representation which is closer to the practical applications. When multiple things might happen at the same time, it is necessary to take into account the duration and the concurrency of the actions. Temporal planning is an approach to automated planning which takes into account these aspects.

Not all of such approaches, however, can rely on effective heuristics. Despite the minor relevance for practical applications, classical planning is remained the most predominant field and, as a consequence, the one that has led to the greatest results. For other approaches, however, either heuristics are unavailable or they result to be not much effective in guiding the resolution process. This thesis aims at reducing the performance gap between two of the existing approaches at domain-independent planning by introducing new heuristics for one of them. In particular, the thesis aims at reproducing some of the results which have been successfully applied in classical planning in another approach to planning, called timeline-based, which is better suited at addressing real-world applications, yet, till now, less efficient from a performance perspective.

The overall goal of this thesis is to improve the efficiency of timeline-based reasoners, possibly taking inspiration from techniques applied in other approaches to planning. To this end, Chapter 2 introduces, from a formal perspective, what automated planning is and some of the most relevant approaches at solving automated planning problems. Chapter 3 describes a first attempt at narrowing the performance gap between classical and timeline-based planning. Since the obtained results were not satisfactory enough, two new heuristics have been proposed. The implementation of such a heuristics, however, required a general reorganization of the existing solver architecture. While Chapter 4 introduces the data structures which allowed the reorganization of the architecture, Chapter 5 introduces the reorganization of the solver architecture, which has led to the development of a new timeline-based solver called ORATIO, and the new heuristics. Chapter 6 describes the RIDDLE language which has been created for describing timeline-based planning problems for the ORATIO solver. Finally, by relying on the same principles introduced in Chapter 3, Chapter 7 attempts at going “beyond” automated planning, showing how it is possible to integrate, in a whole, different forms of reasoning.
An Introduction to Automated Planning and some of its Challenges

This chapter introduces, from a formal perspective, what automated planning is and three different approaches at solving automated planning problems: the classic approach, the partial-order approach and the timeline-based one. Since the main objective of this thesis consists in improving the performance of timeline-based planners through the introduction of effective heuristics, it is important to have a good understanding of the classic approach, i.e., the one in which the most effective heuristics have been developed. This will lay the foundation for understanding classic heuristics and the characteristics of classical planning that allow such heuristics to significantly increase the performance of solvers. Before introducing the timeline-based approach, it might be interesting to introduce an approach at solving classical planning problems called partial-order based. Although this approach aims at solving the same class of problems of the classical approach, it does so in a very different way, losing those features which allow to the classic heuristics to be effective. Partial-order planning, however, possesses some characteristics that, more than other approaches, allow reasoning on temporal aspects. For this reason, it is one of the most used approaches in temporal planning. Additionally, as will be shown soon, this approach shares some commonalities with the timeline-based approach, hence representing a good bridging point between the classical approach and the timeline-based one.

Regardless of the chosen approach, a typical way to introduce the reader to automatic planning consists in relying on a conceptual model, i.e., an abstraction for describing in a simple manner the main elements of a problem, called state-transition systems [34]. Although, as will be shown soon, it can significantly depart from the computational aspects for solving the problem, this conceptual model can be very useful for explaining the underlying basic concepts.

Formally, a state-transition system is a 4-tuple $\Sigma = (S, A, E, \gamma)$, where:
- $S = \{s_1, s_2, \ldots\}$ is a finite or recursively enumerable set of states;
- $A = \{a_1, a_2, \ldots\}$ is a finite or recursively enumerable set of actions;
– $E = \{e_1, e_2, \ldots\}$ is a finite or recursively enumerable set of events; and
– $\gamma = S \times (A \cup E) \rightarrow 2^S$ is a state-transition function.

Figure 2.1 represents a state transition system with states $S = \{s_1, \ldots, s_7\}$, actions $A = \{a_1, \ldots, a_5\}$, events $E = \{e_1, e_2, e_3\}$ and a state transition function $\gamma$ such that, for example, returns state $s_4$ if applying action $a_1$ in state $s_1$ or, similarly, state $s_6$ if event $e_2$ occurs in state $s_7$.

It is worth to notice that both actions and events contribute to the evolution of the system. However, while actions are controlled by the agent, events represent contingent transitions which, although might be taken into account by the agent, are not controllable by it.

Given a state transition system $\Sigma$, the purpose of automated planning is to find which actions to apply to which states in order to achieve some objective when starting from a given situation. A plan is, indeed, a structure characterized by such appropriate actions. The objective can be specified in several different ways:

– The simplest specification consists of a goal state $s_g$ or, more in general, a set of goal states $S_g$. In this case, the objective is achieved by any sequence of state transitions that ends at one of the goal states.

– More generally, the objective might consist in satisfying some conditions over the sequence of states followed by the system. For example, one might want to require states to be avoided, states that the system should reach at some point, and states in which it should stay for some time.

– Another alternative is to specify the objective as the tasks that the system should perform. These tasks can be defined recursively, for example, as sets of actions and other tasks.

It is worth highlighting that the above concept of what a plan is, is shared among the different approaches at automated planning. Such approaches, however, differ mainly on how the states, actions and events, are represented as well as for the different capabilities in reasoning about objectives. Before moving further in describing some of these approaches, however, it is worth introducing some basic terminology related to formal logic. Such terminology, indeed, is often shared among the different approaches and is therefore preparatory to the understanding of the topics that will be dealt with later.

In formal logic [11] all expressions are composed of constants (e.g., Alice, Bob), variables (e.g., $x$, $y$), predicate symbols (e.g., Father, Married) and function symbols (e.g., age, distance). The difference between predicates and functions is that predicates take on values of True or False, whereas functions may take on any constant as their value. It is customary to use lowercase symbols for variables and functions and capitalized symbols for constants and predicates.

From these symbols, expressions are built up as follows: a term is any constant, any variable, or any function applied to any term (e.g., Alice, $x$, age(Bob)). A literal is any predicate, or its negation, applied to any term (e.g., Father(Alice,Bob),
Figure 2.1: A state transition system example.

$\neg$GreaterThan ($distance(A, B), 120$)). If a literal contains a negation ($\neg$) symbol, we call it negative literal, otherwise we call it positive literal. Such a characteristic is also called the polarity of the literal.

A clause is any disjunction of literals. More specifically, a Horn clause is a clause having at most one positive literal, such as

$$H \lor \neg L_1 \lor \ldots \lor \neg L_n$$

where $H$ is the positive literal, and $\neg L_1 \lor \ldots \lor \neg L_n$ are negative literals. Since $(B \lor \neg A) = (B \leftarrow A)$ and $\neg (A \lor B) = (\neg A \lor \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \land \ldots \land L_n)$$

Whatever the notation, the Horn clause preconditions $L_1 \land \ldots \land L_n$ are called the clause’s body or, alternatively, the clause’s antecedents. The literal $H$ that forms the postcondition is called the clause’s head or, alternatively, the clause’s consequent. Additionally, a Horn clause with exactly one positive literal is called a definite clause; a definite clause with no negative literals is called a fact; and a Horn clause without the positive literal is called a goal clause.

Finally, in the non-propositional case, all variables in a Horn clause are implicitly universally quantified with the scope being the entire clause. Thus, for example:

$$\neg Human(x) \lor Mortal(x)$$

stands for

$$\forall x (\neg Human(x) \lor Mortal(x))$$

which is logically equivalent to

$$\forall x (Mortal(x) \leftarrow Human(x))$$
Once described part of the basic formalism, it is possible to introduce some of the most common approaches to automated planning. As we will see, some of these approaches will require the introduction of further formalisms. For the moment, however, it will be provided a glimpse of what automated planning is through its simplest (yet, most studied) approach.

2.1 Classical planning

This section describes classical planning\(^1\), an approach to automated planning which makes use of mathematical logic for representing and reasoning about knowledge. Historically, this approach began with the early works on GPS [86] and on situation calculus [79] and continued for one of the most influential works on automated planning: STRIPS [43]. It is worth to notice, indeed, that the high expressiveness of the state transition system $\Sigma$, as introduced in the previous section, has to cope with computational complexity and efficiency issues. When applied to some realistic scenarios, indeed, the number of states can be extraordinarily huge. Searching for a path, within the system, that allows the achievement of prefixed objectives could be non-trivial. Additionally, dealing with uncertainty coming from the non-determinism of the environment, might result to be extremely challenging. For these reasons, in the past, it has been often chosen to simplify the model. A restricted state-transition system is one that meets some restrictive assumptions on the characteristics of the state transition system $\Sigma$. The typical restrictions are the following ones:

Finite set of states. The system $\Sigma$ has a finite set of states.

Fully observable states. The system $\Sigma$ is fully observable, i.e., one has complete knowledge about the states of $\Sigma$.

Deterministic states. The system $\Sigma$ is deterministic, i.e., for every state $s$ and for every event or action $u$, $|\gamma(s,u)| \leq 1$. If an action is applicable to a state, its application brings a deterministic system to a single other state. In other words, applying an action is always successful and its outcomes are always predictable, regardless of any possible uncertainty. It is worth recalling that this is not always obvious since, for example, a robot might fail in grasping some object or, in case of rolling a dice, the outcome of an action might not be predictable.

Static states. The system $\Sigma$ is static, i.e., the set of events $E$ is empty. $\Sigma$ has no internal dynamics; it persists in the same state until the controller applies some action.

Restricted goals. Goals are specified either as an explicit goal state $s_g$ or as a set of goal states $S_g$, therefore, we are aiming at any sequence of state transitions that ends at

\(^{1}\)This class of planning problems is also referred to in the literature as STRIPS planning, in reference to STRIPS, an early planner for restricted state-transition systems [43].
any of the goal states. Extended goals such as states to be avoided and constraints on state trajectories or utility functions are not handled under this restricted assumption.

### Sequential plans

A solution plan to a planning problem is a linearly ordered finite sequence of actions.

### Implicit time

Actions and events have no duration; they are instantaneous state transitions. This assumption is embedded in state-transition systems since time is not represented explicitly.

### Offline planning

The reasoning process is not concerned with any change that may occur in $\Sigma$ while it is happening, focusing on the given initial and goal states regardless of the current dynamics, if any.

In case all of these simplifications apply, planning can be reduced to the following problem:

Given $\Sigma = (S,A,\gamma)$, an initial state $s_0$ and a set of goal states $S_g$, find a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$ corresponding to a sequence of states $\langle s_0, s_1, \ldots, s_k \rangle$ such that $s_1 \in \gamma(s_0, a_1), s_2 \in \gamma(s_1, a_2), \ldots, s_k \in \gamma(s_{k-1}, a_k)$ and $s_k \in S_g$.

Classical planning refers generically to planning for this kind of restricted state-transition systems. Although classical planning is not the unique possible relaxation, many others have been, indeed, proposed, it is worth talking about it since many heuristics have been developed for efficiently solving these kinds of problems, allowing significant performance improvements. Such heuristics, as we will see, will be a foundation for the development of new heuristics that will allow us to increase the performance of reacher models such those based on timelines. In the following, some formal definitions and some hints on how to solve these problems are given.

### Definition 1

Let $L = \{p_1, \ldots, p_n\}$ be a finite set of proposition symbols. A planning domain on $L$ is a restricted state-transition system $\Sigma = (S,A,\gamma)$ such that:

- $S \subseteq 2^L$, i.e., each state $s$ is a subset of $L$. Intuitively, $s$ tells us which propositions currently hold. If $p \in s$, then $p$ holds in the state of the world represented by $s$, and if $p \notin s$, then $p$ does not hold in the state of the world represented by $s$.

- Each action $a \in A$ is a triple of subsets of $L$, which we will write as $a = (\text{precond}(a), \text{effects}^+(a), \text{effects}^-(a))$. The set $\text{precond}(a)$ is called the preconditions of $a$, and the sets $\text{effects}^+(a)$ and $\text{effects}^-(a)$ are called, respectively, the positive and negative effects of $a$. We require these two sets of effects to be disjoint, i.e., $\text{effects}^+(a) \cap \text{effects}^-(a) = \emptyset$. The action $a$ is applicable to a state $s$ if $\text{precond}(a) \subseteq s$.

- $S$ has the property that if $s \in S$, then, for every action $a$ that is applicable to $s$, the set $(s \setminus \text{effects}^-(a)) \cup \text{effects}^+(a) \in S$. In other words, whenever an action is applicable to a state, it produces another state. This is useful to us because once we know what $A$ is, we can specify $S$ by giving just a few of its states.
The state-transition function is \( \gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a) \) if \( a \in A \) is applicable to \( s \in S \), and \( \gamma(s, a) \) is undefined otherwise.

The above definition establishes one of the possible ways to represent the states and, consequently, the actions of \( \Sigma \). Specifically, states are represented by a set of propositions which are either true or false. Actions can be applied in a given state if some of such propositions (i.e., the preconditions of the action) are true in the state. Finally, the outcome of an action consists in making some of the propositions (i.e., the positive effects) true and some others (i.e., the negative effects) false. Figure 2.2 shows a synthetic example of such a restricted state transition system. In the example, \( L = \{ p_1, p_2, p_3 \} \) so the size of \( S \) is at most \( 2^3 = 8 \). The set of actions \( A = \{ a_1, a_2, a_3, a_4 \} \) is such that both \( a_1 \) and \( a_2 \), having the set of propositions \( \{ p_1, p_2 \} \) as preconditions, are applicable both in the \( s_1 \) and \( s_4 \) states. However, while applying \( a_1 \) in the \( s_1 \) results in the addition of \( p_3 \) and in the removal of \( p_1 \), hence in the state \( s_2 \), applying \( a_2 \) in the same state results in the addition of \( p_3 \) and in the removal of \( p_2 \), hence in the state \( s_3 \).

By exploiting propositions, it is similarly possible to define a planning problem.

Definition 2. A planning problem is a triple \( \mathcal{P} = (\Sigma, s_0, g) \), where:

- \( s_0 \), the initial state, is a member of \( S \).

- \( g \subseteq L \) is a set of propositions called goal propositions that give the requirements that a state must satisfy in order to be a goal state. The set of goal states is \( S_g = \{ s \in S \mid g \subseteq s \} \).

Specifically, the initial state is described, like other states, by means of a set of true and false propositions\(^2\). A set of propositions, however, is used to describe the

\(^2\)Usually, relying on the closed world assumption, those propositions which are not expressed as true are by default considered as false.
set of goal states. In particular, any state in which the goal propositions are true is, by
definition, a goal state. Following the example of Figure 2.2, the initial state $s_0$ can be,
for instance, $s_1$, and is hence described by $s_0 = \{p_1, p_2, \neg p_3\}$. Suppose, however,
the set of goal propositions is $g = \{p_1, p_3\}$, the set of goal states is given by $S_g = \{s_3, s_4\}$
(i.e., all of which the propositions $p_1$ and $p_3$ concurrently hold).

The following definition formally describes what a plan is.

**Definition 3.** A plan is any sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$, where $k \geq 0$. The
length of the plan is given by $|\pi| = k$ the number of its actions.

The state produced by applying the plan $\pi$ to a state $s$ is the state that is produced
by applying the actions of $\pi$ in the given order. The state-transition function $\gamma$ can be
extended as follows:

$$
\gamma(s, \pi) = \begin{cases}
  s & \text{if } k = 0 \text{ (i.e., } \pi \text{ is empty)} \\
  \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_k \rangle) & \text{if } k > 0 \text{ and } a_1 \text{ is applicable to } s \\
  \text{undefined} & \text{otherwise}
\end{cases}
$$

**Definition 4.** Let $\mathcal{P} = (\Sigma, s_0, g)$ be a planning problem. A plan $\pi$ is a solution for $\mathcal{P}$ if $g \subseteq \gamma(s_0, \pi)$.

As an example, a possible solution for a planning problem having the state transition
system of Figure 2.2, the initial state $s_1$ and the goal propositions $g = \{p_1, p_3\}$ is
$\pi = \langle a_1, a_3, a_3, a_2 \rangle$. Applying such a plan to the initial state $s_1$, indeed, would result
in the state $s_3$ in which the goal propositions hold.

It is worth noting that most of the common formalisms (e.g., [43, 56, 45]) shrink the
representation of the state by exploiting predicate symbols. Although such predicates
do not make the formalism more expressive, the description of the domain models,
especially regarding the actions, becomes more compact and readable. For example,
instead of using the propositions $at\_room_0$ and $at\_room_1$, for representing the position
of a robot, it is possible to use a predicate $At(\?x)$ whose parameter $\?x \in \{room_0, room_1\}$
represents all the possible locations of the robot. Similarly, different actions, differing
in the value of certain parameters, can be grouped together in operators. For example,
the operator $GoTo(\?x)$, representing a robot going to a given location $x$, can be used to
compactly describe the two actions $GoTo(room_0)$ and $GoTo(room_1)$.

Classical planning introduces restrictions to the representation of state transition
system which, in some cases, may be too steep. For this reason, through the years, it has
incorporated an increasing number of extensions including durative-actions and
numeric fluents [45, 53], derived predicates and timed initial literals [41], state-trajectory
constraints and preferences [55] and object-fluents. It is not surprising, being the fore-
runner, that representing the state by means of propositions constitutes one of the most
widely used approaches [88, 96, 58].

---

3 A common solution evaluation criteria consists in measuring the length of the plan (also called makespan). With respect to this criteria, the best plan for the above planning problem would have been $\pi = \langle a_2 \rangle$.

CHAPTER 2. AUTOMATED PLANNING

Figure 2.3: The classic rover domain.

Instead of using propositional atoms, however, it is worth noting that some approaches rely on multi-valued state-variables which, usually, allow a more natural representation of the domain and, in some cases, a significant reduction of the computational complexity. Among such approaches it is worth mentioning the SAS+ formalism [3], on which the well-known Fast Downward planning system [62] and the LAMA planner [92] rely, and the Functional STRIPS one [52].

The classic rover domain. Let us introduce a simple planning problem which might help in understanding concepts. As depicted in Figure 2.3, we have a rover on Mars\(^5\) which has two components: (a) a navigation module, for navigating within the environment, and (b) a pan-tilt unit, for orienting its camera. The state of our domain can be modeled by means of the predicates \(\text{At}(?x)\), whose parameter \(?x \in \{\text{pos}_1, \text{pos}_2, \text{pos}_3, \text{pos}_4\}\) represents the position of the rover, \(\text{PntngAt}(?y)\), whose parameter \(?y \in \{\text{left}, \text{straight}, \text{right}\}\) represents the orientation of the camera, and \(\text{TakenPic}(?x, ?y)\), whose parameters \(?x \in \{\text{pos}_1, \text{pos}_2, \text{pos}_3, \text{pos}_4\}\) and \(?y \in \{\text{left}, \text{straight}, \text{right}\}\) represent pictures taken at position \(?x\) with an orientation of the camera \(?y\). Three operators are introduced for controlling the robot: (i) \(\text{GoTo}(?\text{from}, ?\text{to})\), moving the robot from position \(?\text{from}\) to position \(?\text{to}\), has the set \(\{\text{At}(?\text{from})\}\) as preconditions, the set \(\{\text{At}(?\text{to})\}\) as positive effects and the set \(\{\text{At}(?\text{from})\}\) as negative effects; (ii) \(\text{LookAt}(?\text{from}, ?\text{to})\), turning the camera from orientation \(?\text{from}\) to orientation \(?\text{to}\), has the set \(\{\text{PntngAt}(?\text{from})\}\) as preconditions, the set \(\{\text{PntngAt}(?\text{to})\}\) as positive effects and the set \(\{\text{PntngAt}(?\text{from})\}\) as negative effects; and (iii) \(\text{TakePicture}(?\text{at}, ?\text{pntng})\), taking a picture, has the set \(\{\text{At}(?\text{at}), \text{PntngAt}(?\text{pntng})\}\) as preconditions, the set \(\{\text{TakenPic}(?\text{at}, ?\text{pntng})\}\) as positive effects and an empty set as negative effects. Initially, the rover is at position \(\text{pos}_2\) and its camera is looking straight forward. The initial state is hence described by the set of propositions \(\{\text{At}(\text{pos}_2), \text{PntngAt}(\text{straight})\}\). Finally, we want to take a picture at position \(\text{pos}_3\) while pointing at right, hence, the set of goal propositions is \(\{\text{TakenPic}(\text{pos}_3, \text{right})\}\).

\(^5\)This problem is a simplification of the GOAC problem as described in [49].
\(^6\)Notice that, by relying on the closed world assumption, the propositions \(\{\neg\text{At}(\text{pos}_1), \neg\text{At}(\text{pos}_3), \ldots\}\), also, hold in the initial state.
2.1. CLASSICAL PLANNING

Solving classical planning problems

Now that the basic ingredients have been introduced, some hint on how to solve classical planning problems will be provided. To begin, the simplest classical planning algorithms are state-space search algorithms. These are search algorithms in which the search space is a subset of the state space: each node corresponds to a state of the world, each arc corresponds to a state transition, and each plan corresponds to a path in the state transition system starting from the initial state. Two simple algorithms that solve the classical planning problem will be informally presented: (a) a first algorithm that searches forward from the initial state of the world trying to find a state that satisfies the goal formula and (b) a second algorithm that searches backward from the goal formula trying to find the initial state. It is worth noting that both the algorithms, as well as all those following in this chapter, are presented as non-deterministic, i.e., they involve non-deterministic choices, for exposure purposes. Each non-deterministic choice, however, might be seen as the spawning a set of parallel CPUs which, in parallel, proceed according to their own execution traces. The set of execution traces can be represented as an execution tree in which each node represents an execution trace, till a non-deterministic choice is taken, and its children represent the subprocesses spawned by the non-deterministic choice. This tree is called the procedure’s search tree or search space. Any practical implementation, however, must be deterministic, hence, must have a control strategy for visiting the nodes of the tree. Any tree traversal strategy like depth-first, breadth-first, best-first, A*, etc. can, in principle, be exploited for making the following algorithms deterministic, however, choosing the right one might strongly affect the efficiency of the algorithm.

Forward search

One of the simplest planning algorithms for solving classical planning problems is the FORWARD-SEARCH algorithm. The idea is, intuitively, to start from the initial state applying actions until a state in which the goal propositions are satisfied. It is worth noting that since we envision uses in which the number of actions applicable to any given state might be quite large, such a simple system would generate an undesirably large tree of states and would thus result to be impractical [43]. It is worthwhile, in any case, to introduce this approach for didactic, as well as historical reasons.

A non-deterministic version of this algorithm is shown through the pseudo-code in Figure 2.4. The algorithm takes as input the planning problem \( P = (\Sigma, s_0, g) \). If \( P \) is solvable, then FORWARD-SEARCH \( (\Sigma, s_0, g) \) returns a solution plan; otherwise it returns \( \bot \). After initializing the \( s \) variable with the initial state and the plan \( \pi \) with the empty plan, the algorithm enters into the main solving loop. In case \( s \) satisfies \( g \) (i.e., all the propositions in \( g \) are true in \( s \)) the \( \pi \) plan is returned, otherwise, the algorithm collects all the actions which are applicable in the state \( s \) and, for each of them, creates a branch and, non-deterministically, chooses an action \( a \), applies \( a \) in \( s \), replacing \( s \) with the state resulting from the application of the action and, finally, adds \( a \) to the current plan \( \pi \).
procedure `FORWARD-SEARCH(Σ, s₀, g)`

\[ s ← s₀ \]
\[ π ← \text{the empty plan} \]

loop

if \( s \) satisfies \( g \) then return \( π \)
\[ \text{applicable} ← \{ a \mid a \text{ is an action in } Σ, \text{ and } \text{precond}(a) \text{ is true in } s \} \]
if \( \text{applicable} = \emptyset \) then return \( ⊥ \)

non-deterministically choose an action \( a \in \text{applicable} \)
\[ s ← γ(s, a) \]
\[ π ← π. a \]

Figure 2.4: A non-deterministic forward search planning algorithm.

**Solving the classic rover domain through forward search.** In the classic rover domain many actions (i.e., `GoTo(pos2, pos1)`, `GoTo(pos2, pos3)`, `GoTo(pos2, pos4)`, `PntngAt(straight, left)`, `PntngAt(straight, right)` and `TakePicture(pos3, straight)` etc.) are applicable in the initial state. The algorithm has, hence, to non-deterministically choose an action between them. Applying the action `GoTo(pos2, pos3)`, for example, results in the state described by the propositions \( \{\text{At}(pos3), \text{PntngAt}(\text{straight})\} \). Finally, by applying the actions `LookAt(straight, right)` and, subsequently, the newly become applicable action `TakePicture(pos3, right)`, results in the state described by the propositions \( \{\text{At}(pos3), \text{PntngAt}(\text{right}), \text{TakenPic}(pos3, \text{right})\} \) in which the goal propositions (i.e., `TakenPic(pos3, right)` holds. As consequence, the plan \( π = \langle \text{GoTo(pos2, pos3)}, \text{LookAt(straight, right)}, \text{TakePicture(pos3, right)} \rangle \) represents a solution plan.

**Backward search**

Conversely to the forward case, planning can also be done using a backward search. The idea is to start from the goal and apply inverses of the actions to produce subgoals, stopping if we produce a set of subgoals satisfied by the initial state. The set of all states that are predecessors of states in \( S_g \) is:

\[ \Gamma^{-1}(g) = \{ s \mid \text{there is an action } a \text{ such that } γ^{-1}(g, a) \text{ satisfies } g \} \]

This is the basis of the `BACKWARD-SEARCH` algorithm and it is shown through pseudo-code in a non-deterministic version in Figure 2.5. After initializing the plan \( π \) with the empty plan, the algorithm enters into the main solving loop. The algorithm selects all the actions which are relevant for achieving the goal propositions \( g \) and, for each of them, creates a branch. Subsequently, an action \( a \) is non-deterministically chosen, it is added to (the left of) the current plan \( π \) and the set of goal propositions is updated by removing the positive effects of \( a \) and by adding the preconditions of \( a \).

Choosing between the forward and the backward approaches can be guided by the topology of the restricted state transition system. In case the number of outgoing edges

\[ ^7 \text{Again, by relying on the closed world assumption, the propositions } \{\neg\text{At}(pos1), \neg\text{At}(pos2), \ldots\}, \text{ also, hold in the resulting state.} \]
2.1. CLASSICAL PLANNING

<table>
<thead>
<tr>
<th>procedure BACKWARD-SEARCH(Σ, s₀, g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi \leftarrow ) the empty plan</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>( \text{if } s₀ \text{ satisfies } g \text{ then return } \pi )</td>
</tr>
<tr>
<td>relevant ( \leftarrow {a \mid a \text{ is an action in } \Sigma \text{ that is relevant for } g} )</td>
</tr>
<tr>
<td>( \text{if relevant } = \emptyset \text{ then return } \bot )</td>
</tr>
<tr>
<td>non-deterministically choose an action ( a \in \text{relevant} )</td>
</tr>
<tr>
<td>( \pi \leftarrow a.\pi )</td>
</tr>
<tr>
<td>( g \leftarrow \gamma^{-1}(g, a) )</td>
</tr>
</tbody>
</table>

Figure 2.5: A non-deterministic backward search planning algorithm.

in the system greatly exceeds the number of incoming edges, for example, it might be preferable the backward approach since the branching factor is greatly reduced. Conversely, in case the number of incoming edges greatly exceeds the number of outgoing edges, for the same reason, the forward approach is preferable. Finally, in the degenerate case in which the number of incoming (outgoing) edges is at most one, problems can be easily solved in polynomial time (i.e., without the need of searching for a solution) through the backward (forward) approach.

Solving the classic rover domain through backward search. In the classic rover case only TakePicture(\text{pos}_3, \text{right}) is relevant for achieving the goal propositions. By applying this action backward, the proposition TakenPic(\text{pos}_3, \text{right}) is removed from \( g \) while the set \{\text{At}(\text{pos}_3), \text{PntngAt(right)}\} is added to \( g \). Relevant actions are now, for example, GoTo(\text{pos}_2, \text{pos}_3), which removes the proposition \text{At}(\text{pos}_3)\) from \( g \) and adds the proposition \text{At}(\text{pos}_3)\) to it. Finally, by applying LookAt(\text{straight}, \text{right}), the proposition \text{PntngAt(right)}\) is removed from \( g \) while \text{PntngAt(straight)}\) is added to \( g \) resulting in a set of goal propositions which are satisfied in \( g_0 \).

2.1.1 Heuristics for classical planning

Despite the restrictions imposed by classical planning, the search space might become exponentially large (see [12] for a computational complexity analysis of classical planning). As a consequence, by relying on some heuristics, most of the planning algorithms attempt to find a solution without exploring the entire search space. Finding domain-independent heuristics for planning is a research field which has been inaugurated in [10] by introducing the \( h_{\text{add}} \) and the \( h_{\text{max}} \) heuristics described in this section. Since the introduction of such heuristics, furthermore, many heuristics have been proposed mostly relying on delete-relaxation, like the \( h^{FF} \) heuristic [66] and the causal graph heuristics [62], on landmarks, like in [67, 89], on the critical path, like the \( h^n \) heuristic [61, 60] or, lastly, on abstraction, like in [40] or in [63, 64]. It is worth anticipating that the heuristics that allow to efficiently solve timeline-based planning problems, described later on in this thesis (Section 5), are strongly inspired by the \( h_{\text{add}} \) and the \( h_{\text{max}} \) heuristics.
Before going into details of the \( h_{\text{add}} \) and the \( h_{\text{max}} \) heuristics, some general basic concepts about heuristics are provided. To make informed guesses about which choices are more likely to lead to a solution, indeed, it is common to use a heuristic function \( h : s \rightarrow \mathbb{R} \) that returns an estimate of the minimum cost of getting from a state \( s \) to a state satisfying \( g \). Specifically, this heuristic function can be defined as:

\[
h(s) \approx h^*(s) = \min \{ \text{cost}(\pi) \mid \gamma(s, \pi) \text{ satisfies } g \}
\]

where \( h^*(s) \) represents the real, unknown, cost to reach a goal from the state \( s \).

In case the heuristic function never overestimates the \( h^*(s) \) cost of reaching the goal state, i.e., the estimated cost from the current point in the path to a solution is never higher than the lowest possible cost (formally, \( 0 \leq h(s) \leq h^*(s) \)), the heuristic function is said to be admissible.

Clearly, the computational effort for building a heuristic function might be considered worthwhile whenever the function can be computed in a lower (possibly, polynomial) amount of time than directly finding the solution, providing a (possibly, exponential) reduction in the number of nodes searched by the planning algorithm. As an example, the simplest possible heuristic function is \( h_0(s) = 0 \) for every state \( s \). This heuristic is admissible and trivial to compute. Unfortunately, it provides no useful information. We usually would want a heuristic function which would provide a better estimate of the costs for solving our problems.

The best-known way for obtaining heuristic functions is relaxation. Specifically, given a restricted state-transition system \( \Sigma = (S, A, \gamma) \) and a planning problem \( \mathcal{P} = (\Sigma, s_0, g) \), relaxing means weakening some of the constraints that restrict what the states, actions, and plans are, when an action or a plan is applicable and/or what goals it achieves. A very intuitive relaxation technique for the classical planning problems, for example, is to completely ignore the effects\( -\)\( (a) \) list of the actions. Automated planning literature refers to this relaxation as delete relaxation\(^8\). Many heuristics are based on this relaxation.

The following sections present some of the most popular heuristics that allowed considerable increases in performance in solving classical planning problems.

**The \( h_{\text{add}} \) heuristic**

The \( h_{\text{add}} \) heuristic (see [10]) is one of the first heuristics developed with the aim of efficiently solving classical planning problems. According to this heuristic, both the initial state and the actions can be exploited for defining a graph in the propositions space in which for every action \( a \) there is a directed link from the preconditions of \( a \) to its positive effects. The cost of achieving a proposition \( p \) is then reflected in the length of the paths that lead to \( p \) from the initial state.

**Definition 5.** Let \( s \in S \) be a state, \( p \) a proposition, and \( g \) a set of propositions. The minimum distance from \( s \) to \( p \), denoted by \( \Delta^*(s, p) \) is the minimum number of actions required to reach, from the state \( s \), a state containing \( p \). The minimum distance from \( s \) to \( g \), \( \Delta^*(s, g) \), is the minimum number of actions required to reach, from the state \( s \), a state containing all the propositions in \( g \).

\(^8\)The effects\( -\)\( (a) \) list is also called delete list of the action \( a \).
2.1. CLASSICAL PLANNING

The \( h_{\text{add}} \) heuristic represents one of the first attempts to estimate this minimum distance. The heuristic, indeed, ignores the effects \( -^+ (a) \) list of all the actions and approximates the distance to \( g \) as the sum of the distances to the propositions in \( g \). The estimate \( \Delta_{\text{add}} \) is then given by the following equations:

\[
\begin{align*}
\Delta_{\text{add}} (s, p) &= 0 & \text{if } p \in s \\
\Delta_{\text{add}} (s, g) &= 0 & \text{if } g \subseteq s \\
\Delta_{\text{add}} (s, p) &= \infty & \text{if } \forall a \in A, p \notin \text{effects}^+ (a) \\
\end{align*}
\]

otherwise:

\[
\begin{align*}
\Delta_{\text{add}} (s, p) &= \min_a \{ 1 + \Delta_{\text{add}} (s, \text{precond} (a)) \mid p \in \text{effects}^+ (a) \} \\
\Delta_{\text{add}} (s, g) &= \sum_{p \in g} \Delta_{\text{add}} (s, p)
\end{align*}
\]

As mentioned earlier, these equations give an estimate of the distance from \( s \) to \( g \) in the relaxed problem and might be used to guide the search of the unrelaxed problem. The first two equations are simple termination conditions. The third one says that \( p \) is not reachable from \( s \) if the domain contains no action that produces \( p \). The fourth equation defines \( \Delta_0 (s, p) \) recursively with respect to \( \Delta_0 (s, g) \). Finally, the last equation states that the distance to \( g \) is the sum of the distances to its propositions. Note that these formulas do ignore the negative effects of the actions. Furthermore, the relaxation intuition, called independence relaxation, that each proposition in \( g \) can be reached independently of the others, is followed.

We can now define a heuristic function \( h_{\text{add}} (s) \) that gives an estimate of the distance from a node \( s \) to a node that satisfies the goal \( g \) of a planning problem:

\[
h_{\text{add}} (s) = \Delta_{\text{add}} (s, g)
\]

As an example, the estimate \( \Delta_{\text{add}} (s_0, \text{At} (\text{pos} 3)) \), in the case of the classic rover problem described earlier, is 1. On the contrary, the estimate \( \Delta_{\text{add}} (s_0, \text{TakenPic} (\text{pos} 3, \text{right})) \) is 3. Among the possible actions for achieving the proposition \( \text{TakenPic} (\text{pos} 3, \text{right}) \), indeed, the one whose sum of the estimates for achieving its preconditions is minimum is \( \text{TakePicture} (\text{pos} 3, \text{right}) \) since the estimate for achieving the sum of its preconditions is 2.

The \( h_{\text{max}} \) heuristic

The \( h_{\text{add}} \) heuristic is not admissible. In other words, the estimate \( \Delta_{\text{add}} (s, g) \) is not a lower bound on the actual minimal distance \( \Delta^* (s, g) \).

It can be desirable to use admissible heuristic functions for two reasons: (i) we may be interested in getting the shortest plan, or (ii) there may be explicit costs associated with actions and we are using a best-first algorithm for obtaining an optimal (or near optimal) plan. More interestingly, admissible heuristics permit a safe pruning: let \( c \) be the cost (or the length) of a known plan, if \( h (u) > c \), with \( h \) being admissible, then we
are sure that no solution of cost (or length) smaller than $c$ can be obtained from the node $u$. Pruning $u$ is safe in the sense that it does not affect the completeness of the algorithm.

Instead of estimating the distance to a set of propositions $g$ to be the sum of the distances to the elements of $g$, we estimate it to be the maximum distance to its propositions. This leads to an estimate, denoted $\Delta_{\text{max}}$, which is defined by changing the last equation of the $\Delta_{\text{add}}$ estimate to the following:

$$\Delta_{\text{max}}(s, g) = \max \{\Delta_{\text{max}}(s, p) \mid p \in g\}$$

The heuristic function $h_{\text{max}}(s)$ (refer to [10] for further details), which gives an estimate of the distance from a node $s$ to a node that satisfies the goal $g$ of a planning problem, is now defined as:

$$h_{\text{max}}(s) = \Delta_{\text{max}}(s, g)$$

Roughly speaking, the cost for achieving a set of goals $g$ is given by the maximum of the costs of the prepositions in $g$. On the other hand, the cost for achieving a single proposition $p$ is given by the minimum the costs for achieving the preconditions of every action having the proposition $p$ in its positive effects.

As related to the classic rover example, the estimate $\Delta_{\text{add}}(s_0, \text{At}(\text{pos}_3))$ is, again, 1. The estimate $\Delta_{\text{add}}(s_0, \text{TakenPic}(\text{pos}_3, \text{right}))$ is, however, 2. Among the possible actions for achieving the proposition $\text{TakenPic}(\text{pos}_3, \text{right})$, indeed, the one whose sum of maximum estimate for achieving its preconditions is minimum is $\text{TakePicture}(\text{pos}_3, \text{right})$ since the maximum estimate for achieving each of its preconditions is 1.

Popular heuristics are not limited to those mentioned. In [61], for example, the authors notice that the $h_{\text{max}}$ heuristic can be modified to compute costs for pairs of atoms and, once computed for pairs, it can be generalized for triples, quadruples, etc. establishing, actually, a family of heuristics usually known as $h^m$, for $m \geq 1$ (for $m = 1$ $h^m$ behaves equally to $h_{\text{max}}$). These heuristics are, with the growth of $m$, increasingly more informative, yet more costly to compute. As will be shown in Section 5, however, applying the $h^m$ heuristics (for $m > 1$) to the timeline-based case did not prove fruitful since computing the heuristics resulted to be too costly in proportion to the increase in provided information.

### 2.2 Partial-order planning

In the previous section, planning has been considered as the search, either forward or backward, for a path in the graph $\Sigma$ of a state-transition system. Indeed, in the state-space planning case, the search space is given directly by $\Sigma$. This section considers a more elaborate search space which is not $\Sigma$ anymore, but a space in which nodes are partially specified plans and arcs are plan refinement operations intended to further refine a partial plan, i.e., to achieve an open goal or to remove a possible inconsistency. The reason for introducing partial-order planning, as will be shown in details in Section 2.3, consists in its many analogies with the approach we are interested in:
2.2. PARTIAL-ORDER PLANNING

The overall idea underlying the partial-order approach resides in considering search for a solution as two separate operations: (1) the choice of the actions, and (2) the ordering of the chosen actions so as to achieve the desired goals. Finding a solution to a planning problem through this search space, indeed, is typically called plan-space search or partial-order planning. This approach was firstly introduced by the NOAH [97] and by the NONLIN [106] planners, both of which combined plan-space search with hierarchical task refinement, proceeding, some years later, for the SNLP [78] and the UCPOP [88, 112] planners. Compared to classical planning, this approach appears to be more suitable to handle some extensions to the restricted state-transition system (e.g., temporal planning) and, in some cases (see, for example, [4]), might lead to some efficiency gains. Nonetheless, it is undeniably more complex and, consequently, inherently more inefficient than the total-order case.

In order to make up for this loss of efficiency, some works, related to partial-order planning performance, have been proposed investigating aspects of the search control and pruning [54], commitment strategies [81, 82], strategies which leverages state-space planning heuristics [109] and domain features [72]. Additional, it is worth to mention the [87] work which merges most of the above techniques. None of these approaches, however, nowadays, seem to stand the competition. As will be clearer soon, what the partial-order approach misses, compared to the state-space approaches, is a good estimate of the distance between the current state of the solver and a solution state. Such a distance, embodied, for example, by the $h_{add}$ and $h_{max}$ heuristics, might be used to make the search for a solution more efficient. With the loss of an explicit representation of the state, however, it is not clear how to get such an estimate.

Before introducing partial-order planning, however, it is worth introducing some basic formalism about constraint networks on which plan-space search strongly relies. Specifically, the main ingredients of constraint networks are variables and constraints.

**Definition 6.** A variable is an object that has a name and is able to take different values.

A variable (whose name is) $x$ must be given a value from a set that is called the domain of $x$ and is denoted by $\text{dom}(x)$. The domain of a variable $x$ may evolve in time but is always included in a set called initial domain. Depending on the nature of these domains, variables can be distinguished between continuous, having an infinite initial domain usually defined in terms of real intervals, and discrete, whose initial domain contains a finite number of values.

**Definition 7.** A constraint is a restriction on combinations of values that can be taken simultaneously by a set of variables.

A constraint $c$ is defined over a set of variables which constitute the scope of $c$ and are denoted by $\text{scp}(c)$.

A structure composed of variables and constraints is called a constraint network.

**Definition 8.** A constraint network $\mathcal{N}$ is composed of a finite set of variables, denoted by $\text{vars}(\mathcal{N})$, and a finite set of constraints, denoted by $\text{cons}(\mathcal{N})$, such that $\forall c \in \text{cons}(\mathcal{N}), \text{scp}(c) \subseteq \text{vars}(\mathcal{N})$. 
Additionally, an *evaluation* of a constraint network is an assignment of values to some or all the variables. An evaluation is said to be *consistent* if it does not violate any constraint. An evaluation is said to be *complete* if it includes all the variables. Finally, given a constraint network, the problem of finding a consistent and complete evaluation is called Constraint Satisfaction Problem (CSP) (refer to [35, 75] for a comprehensive introduction to CSPs).

The introduction of the previous definitions allows to better describe the main concepts underlying partial-order planning. This approach at planning, indeed, uses a plan structure which is more general than a sequence of actions. Specifically, a plan-space plan is defined as a set of planning operators together with ordering and binding constraints and, in general, it may not correspond to a simple sequence of actions.

**Definition 9.** A partial plan is a tuple $\pi = (A, \prec, B, L)$ where:

- $A = \{a_1, ..., a_k\}$ is a set of partially instantiated actions.
- $\prec$ is a set of ordering constraints on $A$ of the form $a_i \prec a_j$
- $B$ is a set of binding constraints on the variables of the actions in $A$ of the form $x = y$, $x \neq y$ or $x \in D_x$ where $D_x$ is a subset of the domain of $x$.
- $L$ is a set of causal links of the form $a_i \xrightarrow{p} a_j$, such that $a_i$ and $a_j$ are actions in $A$, the constraint $a_i \prec a_j$ is in $\prec$, the proposition $p$ is an effect of $a_i$ and a precondition of $a_j$, and the binding constraints for variables of $a_i$ and $a_j$ appearing in $p$ are in $B$.

The interesting aspect of this approach consists in having the actions only partially (as opposed to fully) instantiated. In other words, the actions’ parameters can be lifted. If on the one hand this allows to significantly reduce the branching factor, given that a large number of actions are considered simultaneously through partially instanced operators, it is also true that, in order to manage these operators, a data structure is needed to handle variables and constraints, hence the introduction of the constraint network.

With this in mind, a plan space is an implicit directed graph whose vertices are partial plans and whose edges correspond to refinement operations. An outgoing edge from a vertex $\pi$ in the plan space is a refinement operation that transforms $\pi$ into a refined partial plan $\pi'$. Intuitively, a refinement operation consists of one or more of the following steps:

- Add an action to $A$
- Add an ordering constraint to $\prec$
- Add a binding constraint to $B$
- Add a causal link to $L$
2.2. PARTIAL-ORDER PLANNING

Plan-space planning is the search in this graph of a path from an initial partial plan denoted \( \pi_0 \) to a node recognized as a solution plan. Since partial plans are defined through actions and their relationships, both goals and the initial state have to be represented into \( \pi_0 \). This can be achieved describing \( \pi_0 \) by means of two dummy actions \( a_0 \), an action with no preconditions and whose effects represent the initial state, and \( a_\infty \), an action with no effects and whose preconditions are the goals \( g \).

Before introducing the planning algorithms in the plan space, however, we need to formally specify what a solution plan in this space is. Plan-space search, indeed, differs from state-space search not only in its search space but also in its definition of a solution plan.

**Definition 10.** A partial plan \( \pi = (A, \prec, B, L) \) is a solution for a planning problem \( P = (\Sigma, s_0, g) \) if:

- its ordering constraints \( \prec \) and binding constraints \( B \) are consistent
- every sequence of totally ordered and totally instantiated actions of \( A \) satisfying \( \prec \) and \( B \) is a sequence that defines a path in the state-transition system \( \Sigma \) from the initial state \( s_0 \) corresponding to effects of action \( a_0 \) to a state containing all goal propositions in \( g \) given by preconditions of \( a_\infty \).

Unfortunately, this definition does not provide a computable test for verifying plans. In order to establish a resolution procedure for our plan-space planning problem we refer to the concepts of threats and flaws.

**Definition 11.** An action \( a_k \) in a plan \( \pi \) is a threat on a causal link \( a_i \xrightarrow{p} a_j \) iff:

- \( a_k \) has an effect \( \neg q \) that is possibly inconsistent with \( p \), i.e., \( q \) and \( p \) are unifiable;
- the ordering constraints \( a_i \prec a_k \) and \( a_k \prec a_j \) are consistent with \( \prec \);
- the binding constraints for the unification of \( q \) and \( p \) are consistent with \( B \).

**Definition 12.** A flaw in a plan \( \pi = (A, \prec, B, L) \) is either:

- a subgoal, i.e., a precondition of an action in \( A \) without a causal link;
- a threat, i.e., an action that may interfere with a causal link.

These new definitions allow us to redefine the concept of solution plan in a way which is suitable for being exploited by a resolution procedure.

**Proposition 1.** A partial plan \( \pi = (A, \prec, B, L) \) is a solution for a planning problem \( P = (\Sigma, s_0, g) \) if \( \pi \) has no flaws and if the set of ordering constraints \( \prec \) and binding constraints \( B \) is consistent.

Figure 2.6 shows an example of partial plan for the classic rover domain. Filled blocks represent the (partially instantiated) actions with their preconditions above and effects below the box. While solid arrows represent the ordering constraints, dashed arrows represent causal links. It is worth noting that the ?from parameter of the GoTo
action is lifted, indicating a movement of the rover towards a specific direction yet starting from a direction which is still uncertain. Similarly, each action has its own numerical variable representing the time the action applies. Exception made for the $a_0$ action, which is constraint to happen at time zero, also temporal variables are lifted. Finally, it is worth to highlight that all these variables, as well as the constraints among them, are maintained in a constraint network.

The most interesting aspect of the above definitions and propositions, compared to the state-space planning case, resides mainly in the fact that the generated plans allow two different types of flexibility: (i) actions do not have to be totally ordered and (ii) they do not have to be totally instantiated. This remark applies also to action $a_\infty$ therefore partial-order planning allows us to handle partially instantiated goals (e.g., in the classic rover domain it is possible to express goals like $\{\text{TakenPic}(\text{pos}_3, ?pntng\_at)\}$ without the need of instantiating the $?pntng\_at$ variable, aiming at expressing the desire of taking a picture at $\text{pos}_3$ while looking at any direction). These aspects related to flexibility are particularly relevant when tackling plan execution. When dealing with dynamic environments, indeed, failures are not uncommon. Further constraints, for example, might become available at execution time requiring the adaptation of the plan to the actual needs. Additionally, partial-order planning allows to slightly reduce the restrictions of classical planning. As an example, it is much easier to manage constraints about the desired trajectories within the state-transition system $\Sigma$ since it is possible to introduce goals into $\pi_0$ and enforce temporal constraints on them. Finally, partial-order planning allows to express goals in terms of tasks which should be performed by the agent. All of these features, as well as most of the concepts underlying the search algorithms, are in common with timeline-based planning. Before introducing this third approach at automated planning, however, it is worth to provide a typical algorithm for searching in the plan space.

### 2.2.1 An algorithm for partial-order planning

Intuitively, since $\pi$ is a solution when it has no flaw, the main principle is to refine $\pi$, while maintaining $\prec$ and $B$ consistent, until it has no flaw. The basic operations for refining a partial plan $\pi$ toward a solution plan are the following:

- Find the flaws of $\pi$, i.e., its subgoals and its threats.
- Select one such flaw.
- Find ways to resolve it.
2.2. PARTIAL-ORDER PLANNING

\begin{figure}[ht]
\centering
\begin{verbatim}
procedure PSP(π)
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then return π
    select any flaw φ ∈ flaws
    resolvers ← Resolve(φ, π)
    if resolvers = ∅ then return failure
    non-deterministically choose choose a resolver ρ ∈ resolvers
    π' ← Refine(ρ, π)
    return PSP(π')
\end{verbatim}
\caption{The PSP procedure.}
\end{figure}

The procedure PSP(π) is specified in Figure 2.7. It works as follows:

- Choose a resolver for the flaw.
- Refine π according to that resolver.

The PSP procedure is called with an initial plan π₀ and each successful recursion is a refinement of the current plan according to the given resolver. Each invocation of the Resolve procedure introduces new variables and/or constraints to an underlying dynamic constraint network which is responsible for maintaining consistent the domains of the variables with the ≺ and B constraints. Intuitively, refinement operations avoid adding to the partial plan any constraint that is not strictly needed for addressing the refinement purpose (this is called the least commitment principle). It is worth noticing that more than in finding a complete assignment for all the variables of the constraint network, we are interested only in checking its consistency. Because this check should be made for each node of the search space we are interested in efficient polynomial procedures even at the cost of some compromise (i.e., the possibility for false positives in the consistency check procedure). Moreover, since the real world has high degrees of uncertainty, we are interested in maintaining different possible solutions in a single constraint network which might come in handy at plan execution time in order to handle possible unforeseen events.
Solving the classic rover domain. What happen if we apply the partial-order approach at the rover domain? Initially, the \( \pi_0 \) partial plan contains the \( a_0 \) action, at time zero, having \{At\((pos_2)\), PntngAt\((\text{straight})\)\} as positive effects. Additionally, \( \pi_0 \) contains the \( a_{inf} \) action, whose temporal variable has domain, initially, within the \([0, +\infty)\) interval, having \{TakenPic\((pos_3, \text{right})\)\} as preconditions. The OpenGoals procedure, initially, returns the sole \( \text{TakenPic}(pos_3, \text{right}) \) goal which can be resolved by introducing both a partial instantiation of the \( \text{TakePicture}(\?at, \?pntng\_at) \) operator and a causal link linking its effect to the precondition of the \( a_{inf} \) action. It is worth noting that while the introduction of the operator introduces three variables in the underlying constraint network (i.e., the \( \?at \) variable, the \( pntng\_at \) variable and the action’s temporal variable), the introduction of the causal link constraints the \( \?at \) and the \( pntng\_at \) variables to assume, respectively, the \( pos_3 \) and the \( \text{right} \) values. Furthermore, the introduction of the ordering constraint reduces the domain of the \( a_{inf} \) temporal variable to the \([1, +\infty)\) interval. The OpenGoals procedure is called again returning, this time, the \( \text{At}(pos_3) \) and the \( \text{PntngAt}(\text{right}) \) flaws. The first one can be solved by introducing a partial instantiation of the \( \text{GoTo}(\?\text{from}, \?\text{to}) \) operator and a causal link linking its positive effect to the precondition of the earlier introduced \( \text{TakePicture}(\?at, \?pntng\_at) \) action. This leads to the situation depicted in Figure 2.6. The flaws to be solved are, now, \( \text{At}(\?\text{from}) \) and the still unsolved \( \text{PntngAt}(\text{right}) \). While the first one can be resolved by introducing a causal link between the \( \text{At}(pos_2) \) effect of the \( a_0 \) action and the \( \text{At}(\?\text{from}) \) \( \text{GoTo} \)'s precondition, reducing the domain of the \( \?\text{from} \) variable to the sole \( pos_2 \) allowed value, \( \text{PntngAt}(\text{right}) \) can be solved by introducing a new partially instantiated operator, i.e., \( \text{LookAt}(\?\text{from}, \?\text{to}) \), a causal link linking its positive effect and the \( \text{PntngAt}(\text{right}) \) precondition of the \( \text{TakePicture}(\?at, \?pntng\_at) \) action. Finally, a causal link linking the \( \text{PntngAt}(\text{straight}) \) effect of the \( a_0 \) and the precondition of the last introduced action would result in a solution plan. It is worth noting that, conversely to the previously described classical approach, the achieved solution plan does enforce an ordering constraint between the \( \text{GoTo} \) action and the \( \text{LookAt} \) one, hence the gain in flexibility at execution time (and the partial-order name of the approach).

It is worth to highlight the fact that the order in which flaws are processed is not important neither for the soundness nor for the completeness of the procedure. It is, however, very important for efficiency aspects. In this regard, it is worth noting that the notion of explicit states, within the search procedure, is lost. State-space planners receive huge benefit from heuristics which, as has been briefly mentioned in Section 2.1.1, are explicitly defined on the concept of states. Although there are some attempts to generalize state-space heuristics to plan-space planning [87], it turns out that, in general, plan-space planners are not competitive enough with respect to state-space planners on the computationally efficiency field.

2.3 Planning with timelines

Timeline-based planning was first introduced in [85, 84] and, since then, many timeline-based planners have been proposed like, for example, IkTeT [57], EUROPA [69], ASPEN [20], the TRF [48, 14] on which the APSI framework [49] relies and, more recently, PLATINUm [107]. Some theoretical work on timeline-based planning like [47]
was mostly dedicated to explain details of [69], identifying connections with classical planning a-la PDDL [45]. The work on $\text{kiTfT}$ and $\text{Trf}$ has tried to clarify some key underlying principles but mostly succeeded in underscoring the role of time and resource reasoning [17, 73]. The planner CHIMP [104] follows a Meta-CSP approach having meta-Constraints which have only resembles timelines. The Flexible Acting and Planning Environment (FAPE) [39] tightly integrates timelines with acting. The Action Notation Modeling Language (ANML) [101] is an interesting development which combines the HTN decomposition methods with the expressiveness of the timeline representation. Finally, it is worth mentioning that the timeline-based approaches have been often associated to resource managing capabilities. By leveraging on constraint-based approaches, most of the above approaches like $\text{kiTfT}$ [74, 73], [18], [102] or [110] integrate planning and scheduling capabilities. Finally, [22] proposes a recent new formalization of timeline-based planning.

Despite the above systems have been deployed in demanding applications, for example, for controlling autonomous space systems and underwater vehicles, the search control part of these reasoners has always remained significantly unexplored, missing, in fact, the performance boost that has characterized classical planning in recent years. By generalizing the reachability and dependency graphs of state-space planning, [7] represents one of the few works which propose an elaborate heuristic for these representations.

As we have seen in the previous sections, in order to cope with computational complexity, classical planning assumes some strong restrictions on the state transition system $\Sigma$. Partial-order planning aims at relaxing some of the above restrictions paying, however, a cost in terms of efficiency. Timeline-based planning goes even further in this direction. Similarly to the partial-order case, goals can be specified by means of desired trajectories of modeled systems’ subcomponents or, possibly, tasks that such systems should perform. Additionally, the timeline-based approach to planning explicitly addresses temporal aspects. The restrictions about the set of states, about the goals definition, about the implicit representation of time as well as about sequential plans, as a consequence of the richer expressiveness, have been therefore relaxed, resulting in an effective alternative to classical planning for complex domains requiring the use of both temporal reasoning and scheduling features. As expected, however, compared to partial-order planning, performance degrades even more. Before addressing the performance issue, however, it is mandatory to formally introduce what timeline-planning is and how to solve problems in this formalism. Since its first introduction in 1991, however, the concept of timeline-based planning has been reworked several times, adapting it to the specific needs of the moment. The goal of this section is to provide a new definition of timeline-based planning which is as general as possible, so as to include all the characteristics of the timeline-based approaches proposed so far.

In essence, within the timeline-based approaches, planning is addressed by modeling the problem by means of a set of relevant features of the planning domain, called timelines, which need to be controlled in order to obtain a desired temporal behavior. Timelines model entities whose properties may vary in time and which represent one or more physical (or logical) subsystems which are relevant to a given planning context. The planner/scheduler plays the role of the controller for these entities, and reasons in terms of constraints that bound their internal evolutions and the desired properties of
CHAPTER 2. AUTOMATED PLANNING

Figure 2.8: Different types of timelines: a symbolic timeline, a piecewise constant timeline and a continuously changing timeline.

the generated behaviors.

In generic terms, a timeline is a function from time, either continuous or discrete, to a set of values. Formally,

**Definition 13.** A timeline $T$ is a function

$$T : T \rightarrow \mathcal{V}$$

where $T$ is the (either discrete or continuous) domain of time and $\mathcal{V}$ is the (possibly infinite) codomain of the timeline.

According to the modeled subsystem, the codomain $\mathcal{V}$ of a timeline can be either symbolic or numeric. Additionally, in case of numeric timelines, the domain can be either discrete or continuous. Finally, in case of timelines with continuous domains, values can change either piecewise or, according to some (more or less complex) physical law, continuously. Figure 2.8 shows three examples of such types of timelines: a symbolic timeline, a piecewise constant timeline and a continuously changing timeline.

Since the definition of timeline is completely general, it is possible to represent, through these, extremely heterogeneous concepts. There is the need, therefore, of a unifying element that allows to represent contents homogeneously, in a way which is agnostic from the nature of the timeline, so as to allow reasoning. To this end, the concept of token is introduced. Despite the different nature of the timelines, indeed, their values over time are a direct consequence of the tokens that are enforced on them. Without loss of generality, a token is an assertion over a temporal interval. An additional argument is used to indicate the timeline on which the token applies. Formally,

**Definition 14.** A token is an expression of the form:

$$n(x_0, \ldots, x_k) @ [s, e, \tau]$$

where $n$ is a predicate name, $x_0, \ldots, x_k$ are the parameters of the predicate (i.e., constants, numeric variables or symbolic variables), $s$ and $e$ are the temporal parameters of the token (i.e., constants or variables) belonging to $T$ such that $s \leq e$ and $\tau$ is the scope parameter of the token (i.e., a constant or a symbolic variable) representing the timeline on which the token apply.

Roughly speaking, the expression on the left of the “@” symbol represents the assertion while the expression at its right represents the interval. In other words, a token $n(x_0, \ldots, x_k) @ [s, e, \tau]$ asserts that $\forall t$ such that $s \leq t \leq e$, the relation $n(x_0, \ldots, x_k)$ holds at the time $t$ on the timeline $\tau$. How the timelines’ temporal evolutions are extracted by
the tokens enforced on them depends on their specific nature (some relevant examples will be provided later).

A critical aspect to keep in mind is that, in general, tokens’ parameters are variables (see Definition 6) and, as such, can be constrained. In other words, in order to reduce the allowed values for the tokens’ constituting parameters, and thus decreasing the modeled system’s allowed behaviors, it is possible to impose constraints among such variables (and/or between the parameters and other possible variables). Such constraints include temporal constraints, usually expressed by means of interval relations [1], binding constraints between symbolic variables as well as (non)linear constraints among numerical variables (possibly including temporal variables). Finally, it is worth to notice that in a grounded plan each token applies to a specific timeline, reflecting the intuition that tokens describe some aspect of the timeline (i.e., its state or behavior) in time. However, in general, the commitment to a specific timeline may not yet have been made and τ can be treated as any other variable of a constraint network.

Similarly to partial-order planning, the set of tokens and constraints is used to describe the main data structure that will be used to represent nodes of the timeline-based search space: the token network.

**Definition 15.** A token network is a tuple \( \pi = (T, C) \), where:

- \( T = \{t_0, \ldots, t_k\} \) is a set of tokens.
- \( C \) is a set of constraints on the variables of the tokens in \( T \).

\( C \) is required to be consistent, i.e., there exist values for the variables that meet all the constraints. As introduced in Section 2.2 and similarly to is done in partial-order planning, a possible solution for checking the consistency of a token network is to associate it to a constraint network, linking each parameter of each token to a variable of the constraint network and enforcing the constraints \( C \) on them.

Additionally, tokens can be partitioned into two groups: facts and goals. While facts are, by definition, inherently true, goals need to be “achieved”. Specifically, causality, in the timeline-based approach, is defined by means of a set of rules indicating how to achieve goals. Formally,

**Definition 16.** A rule is an expression of the form

\[ n(x_0, \ldots, x_k) @ [s, e, \tau] \leftarrow r \]

where:

- \( n(x_0, \ldots, x_k) @ [s, e, \tau] \) is the head of the rule, i.e., an expression in which \( n \) is a predicate name, \( x_0, \ldots, x_k \) are the parameters of the head (i.e., constants, numeric variables or symbolic variables), \( s \) and \( e \) are the temporal parameters of the head (i.e., constants or variables) belonging to \( T \) such that \( s \leq e \) and \( \tau \) is the scope parameter of the head (i.e., a constant or a symbolic variable) representing the timeline on which the rule apply.
– \( r \) is the body of the rule (or the requirement), i.e., either a slave \( (\text{or target}) \) token, a constraint among tokens (possibly including the \( x_0, \ldots, x_k, s, e, \tau \) variables), a conjunction of requirements or a disjunction of requirements (in the latter case, the disjuncts might be characterized by a cost).

Rules define causal relations that must be complied to in order for a given goal to be achieved. Roughly speaking, rules define how to achieve goals: in order for a goal, having the form of the head of a rule, to be achieved, the body of the rule must be within the plan. It is worth noticing that by combining tokens, constraints, conjunctions and disjunctions, requirements allow to express complex concepts. By assigning costs to disjuncts, additionally, it is possible to assert preferences \([80, 95]\), indicating desires and/or satisfaction levels on the generated plans.

It is worth noticing that tokens included in the body of the rules might represent other goals which might need be achieved too. As a consequence, according to the chosen resolution procedure, other rules might require to be applied, in order to achieve them. Intuitively, the resolution process continues until either a fact is met (which, by definition, is already true), or another goal, which is already known how to achieve, is met. In both cases we talk about unification of the goal with either a fact or another goal.

The last aspect to be addressed is the definition of the timeline-based planning problem which can rely on the above requirement concept.

Definition 17. A timeline-based planning problem is a triple \( P = (\mathcal{T}, \mathcal{R}, r) \), where:

– \( \mathcal{T} \) is a set of timelines.

– \( \mathcal{R} \) is a set of rules.

– \( r \) is a requirement, i.e., either a fact token, a goal token, a constraint among token arguments, a conjunction of requirements or a disjunction of requirements (in the latter case, similar to rules, the disjuncts might be characterized by a cost).

It is worth noting that the above formalism slightly differs from the one usually accepted as standard (i.e., \([20, 47]\) and \([22]\)). In particular, since state-variables are just one of the possible types of timeline, it is preferable, whenever possible, to eliminate from the formalism those peculiarities specifically related to them. According to the author, indeed, there is no reason to include, in the formalism, the transitions, from one value to another, within the same state-variable. Such transitions, whenever needed, can be easily specified through rules while leaving to the user, as well, the possibility to not specify them at all. The definition of the transitions, indeed, implies having the state-variables completely specified for the whole planning horizon. Again, this behavior can easily be enforced through the use of rules, nonetheless, there are cases in which it is preferable to leave the state-variables as partially specified so that during execution, for example, they can be “filled” with values derived from the interaction with the external environment (i.e., exogenous events). The introduction of value transitions, furthermore, associates to each state-variable its own state transition system. Nonetheless, it is not always possible to represent timelines as a sequence of stepwise values.
or, in some cases, it might be cumbersome. Consumable resources, for example, are timelines having a continuous update of their value in time, therefore, value transitions completely lose their meaning. Similarly, reusable resources might be cumbersome in such a formalism. Although such resources are, in principle, representable by means of a state transition system, this system would be completely connected losing, de facto, the usefulness of such a representation. Conversely, in case such resources are modeled by means of continuous (as opposed to discrete) amount of resource usage, the number of states becomes infinite. It is worth noting that also those extensions to classical planning that include numeric fluents, as reported in [53], are not exempt from this issue. The big issue, indeed, might concern the definition of timeline which both in [20], in [47] and [22] is probably faulty (or, at least, counter intuitive): although state-variables are timelines, the opposite is not necessarily true. Similarly to state-variables, resources represent another kind of timeline. In other words, state-variables are just a subset of the possible timelines and, in principle, many others can be defined.

Although the change might seem minor, compared to previous formalisms, there is also an inversion in the direction of the arrow in the rule definition (which, in other formalisms, are also called synchronizations or compatibilities). Goals are objectives which, by definition, require to be achieved. Rules tell how to achieve goals: the body of a rule specifies what is needed for a goal, of the kind specified in the head of the rule, to be achieved. Although it may seem a mere stylistic effort, this inversion deeply modifies the approach that a potential user has with the formalism, paving the way, additionally, for applying classical planning heuristics to the timeline-based case. Despite the appropriate adaptations, indeed, rules can be seen as classical planning operators, having the body of the rule as preconditions and the head of the rule as the sole positive effect. Both the head of the rule and its body, however, might involve combinations of numeric variables (e.g., temporal variable) and constraints, making the migration of the heuristics from classical planning to timelines not so straightforward as it might seem at a first step.

The rover domain revised. In order to further clarify the just introduced concepts, it is worth to redesign the rover domain so as to exploit the timeline-based capabilities. Specifically, since the rover is composed of two components, it seems more intuitive, compared to the classical approach, to model the system by means of two timelines representing, respectively, the evolution over time of the navigation module and the evolution over time of the pan-tilt unit. While the allowed values of the navigation timeline can be either \( At(?, ?) \), representing the rover standing at coordinates ?x and ?y, or \( GoingTo(?, ?) \), representing the rover going to coordinates ?x and ?y, the allowed values for the pan-tilt unit can be either \( LookingAt(?pan, ?tilt) \), representing the pan-tilt unit pointing at polar coordinates ?pan and ?tilt, or \( RotatingTo(?pan, ?tilt) \), representing the pan-tilt unit rotating toward the polar coordinates ?pan and ?tilt, or even, \( TakingPic(?x, ?y, ?pan, ?tilt) \), representing the camera taking a picture while the rover is at coordinates ?x and ?y while pointing to the ?pan and ?tilt polar coordinates. The problem would be defined by introducing two fact tokens representing the current state of the system like, for example, \( At(0, 0) \) @ \([0, e, nav]\) and \( PointingAt(0, 0) \) @ \([0, e, pantilt]\), and a goal token like, for example, \( TakingPic(1, 5, 2, 7) \) @ \([s, e, pantilt]\). The left part
of Figure 2.9 shows the allowed values and the possible value transitions within the two timelines. A token $At(\tau_x, \tau_y) @ [s, e, \tau]$ for example, would represent the rover standing at coordinates $\tau_x$ and $\tau_y$ from time $s$ to time $e$, while, given the simplicity of the domain, the $\tau$ variable would represent the sole navigation module timeline. Finally, the right part of Figure 2.9 shows the rules for the rover domain. As an example, in order for a goal represented by the token $At(\tau_x, \tau_y) @ [s, e, \tau]$ to be achieved, either it is already achieved by another token (and, hence, we have an unification of the token with the other token) or the associated rule’s requirement must be present in the current token network. In the latter case, a new $GoingTo(\tau_x, \tau_y) @ [s, e, \tau]$ token called, locally to the rule, $gt$, is added to the token network and its ending time is constrained to be equal to the starting time of the newly achievable goal. The new goal will eventually undergo the same destiny either unifying with another token or another rule would be applied. Finally, it is worth to notice how the $TakingPic$ rule introduces tokens on another timeline (the $nav$ timeline represents the timeline associated to the navigation module) and how the $LookingAt$ rule introduces a disjunction.

2.3.1 Interactions among tokens: the timelines

From a planning perspective, the easiest way to describe a timeline is to consider it as a mere collection of tokens (i.e., those tokens whose $\tau$ parameter assumes, as allowed value, the timeline). The values that the timelines assume over time, as well as the behavior assumed by the planner when new tokens are added to a timeline, depend not only on the tokens and on the modeled domain (i.e., the defined rules), but also on the nature of the timeline itself. Timelines, indeed, introduce further implicit (and higher level) constraints to the set of constraints $C$ of a token network that, similarly to the partial-order planning case, can be explicated through the concept of threat. In the timeline-based case, however, threats might assume different semantics according to the nature of the involved timeline. Typical examples of timelines are multi-valued state-variables [84]. Renewable and consumable resources, like those commonly used in constraint-based scheduling [73], however, also fit within the above definition of timeline.

State-Variables. The state-variable is the most used type of timeline in this approach to planning. Predicates allowed by state-variables are defined by the user during the
2.3.  **PLANNING WITH TIMELINES**

Figure 2.10: Different kinds of timelines.

- **State-Variable:**
  - Definition: For each time instant \( t \in T \), the timeline can assume only one value. This corresponds to a temporally mutual exclusion rule between the tokens on the same state-variable.
  - Management: Overlap of tokens on the same state-variable is resolved by imposing a temporal constraint or a constraint on their \( \tau \) variables.
  - Example: Tokens are applied on different state-variables, and their temporal evolution is given directly by the tokens.

- **Reusable Resource:**
  - Characterized by a resource level \( L : T \to \mathbb{R} \), representing the amount of available resource at any given time.
  - Additional parameters:
    - Max level \( \max \in \mathbb{R} \)
    - Min level \( \min \in \mathbb{R} \)
    - Initial amount \( init \in \mathbb{R} \)
  - Example: Reservoir content is produced when an activity fills it and consumed when it empties.

- **Consumable Resource:**
  - A timeline characterized by producing and consuming activities.
  - Parameters:
    - Resource level \( L : T \to \mathbb{R} \)
    - Max level \( \max \in \mathbb{R} \)
    - Min level \( \min \in \mathbb{R} \)
    - Initial amount \( init \in \mathbb{R} \)
  - Example: Tank refueling and car driving.

Definition of the domain. The semantics of a state-variable is that for each time instant \( t \in T \), the timeline can assume only one value. This corresponds to a temporally mutual exclusion rule between the tokens on the same state-variable.
represents a resource production of amount $a$, and a predicate \textit{Consume}($a$) to represent a resource consumption of amount $a$. Extracting, from the tokens, the temporal evolution of a consumable resource (i.e., the resource level) is more complicated than in the state-variables’ case since it depends on how resource productions and consumptions are placed in time. Additionally, the planner may need to identify an ordering of the involved activities in order to avoid overproductions (resource level $L$ cannot exceed the upper limit $\text{max}$) as well as overconsumption (resource level $L$ cannot be lower than the lower limit $\text{min}$). Clearly, overproductions and overconsumption can be managed as threats raised by consumable resources which, similarly to the state-variable threats, can be resolved either by imposing a temporal constraint for ordering the tokens in time, or by imposing a constraint on their $\tau$ variables indicating on which resource the token applies. Furthermore, constraints on the amounts of the productions and consumptions might also resolve the threat\textsuperscript{9}. Figure 2.10(c) represents an example of consumable resource with its tokens and its profile.

**Reusable Resources.** The last commonly used timeline type, quite popular in the scheduling literature, is the \textit{reusable resource}. An example of reusable resource is the collection of available programmers of an IT company. Such programmers might be assigned to a given project for a specific amount of time after which the resource becomes, again, available. Reusable resources can be modeled as consumable resources whose tokens consume an amount of resource at their start time and produce the same amount at their end time. Rather than a \textit{min} and a \textit{max} level, these timelines are characterized by a \textit{resource capacity} $C \in \mathbb{R}$, representing the physical limit of the available resource. The initial amount of available resource is assumed to be equal to the resource capacity. The sole allowed predicate for reusable resources is \textit{Use}($a$) that represent an instantaneous consumption of resource of amount $a$ at time $s$ and an instantaneous production of resource of amount $a$ at time $e$ for each token assigned to the reusable resource. Let’s assume, for example, that there are two tokens, $t_0$ and $t_1$, assigned to the same reusable resource, such that the constraint $t_0.s < t_1.e \land t_1.s < t_0.e$ holds (this constraint simply forces their overlapping). The expected behavior of the resource is to have a resource usage of $t_0.a$ during $t_0$’s duration when there isn’t overlapping with $t_1$, a resource usage of $t_0.a + t_1.a$ when $t_0$ overlaps with $t_1$, a resource usage of $t_1.a$ during $t_1$’s duration when there is no overlapping with $t_0$ and a resource usage of 0 elsewhere. In other words, reusable resources’ temporal evolution is simply given by the concurrent resource usages. Finally, resource overuses might be managed as threats raised by reusable resources which, similarly to state-variable threats and the consumable resource ones, can be resolved either by imposing a temporal constraint for ordering the tokens in time or by imposing a constraint on their $\tau$ variables indicating on which resource the token applies. Constraints on the amounts of the uses might also resolve the threat. Figure 2.10(b) represents an example of reusable resource with its tokens and its profile.

\textsuperscript{9}It is worth to notice that threats on consumable resources might also be solved through the addition of further productions and consumptions. This aspect makes the reasoning on consumable resources, in general, non-monotonic. A possible workaround, as proposed in [73], consists in \textit{closing} the consumable resource, preventing the introduction of further tokens on the resource, before resolving threats on it.
2.3. PLANNING WITH TIMELINES

It is worth to notice that resources have their own tokens which might be, in principle, unrelated by other state-variables’ tokens. This separation of concepts allows to easily model pure constraint-based scheduling problems without relying on the usage of state-variables. Furthermore, this allows for greater flexibility in constraining tokens of a state-variable with the resources usages (e.g., we can model a resource usage which starts at some given time after an activity represented by a token on a state-variable). Finally, it is worth to highlight that the proposed formalism is open and can easily (in principle) introduce new kinds of timelines (e.g., batteries, whose overcharges represent wastes of energy, classical planning agents, etc.).

2.3.2 Algorithms for timeline-based planning

Despite the differences in representing the state-transition system $\Sigma$, the search space of partial-order planners and timeline-based planners is mostly the same. Specifically, the main difference related to the search space consists in the fact that timeline-based planners generalize the partial-order planning concept of threat including any possible inconsistency which might appear in the timelines. Since there are different kinds of timelines, there might be different types of inconsistencies. Despite the generalization, however, the search space (and, consequently, the solving algorithm) remains substantially unchanged compared to the Algorithm 2.7 for partial-order planning.

Definition 18. A flaw in a token network $\pi = (T, C)$ is either: (i) an open goal (i.e., a token having an undecided value for its $\sigma$ variable), (ii) a threat or (iii) a disjunction.

Similarly to the partial-order case, while flaws can be of different types and can arise for different reasons, what they all have in common is that a search choice is necessary to solve each of them, thus the basic operations for refining a partial plan $\pi$ toward a solution plan are the following:

1. Find the flaws of $\pi$, i.e., its open goals, its threats or its disjunctions.
2. Select one such flaw.
3. Find ways to resolve it.
4. Choose a resolver for the flaw.
5. Refine $\pi$ according to that resolver.

The process proceeds until there is no flaw in $\pi$ and all of its constraints are consistent. In case this condition occurs then $\pi$ is a solution to the planning problem. Similarly to the partial-order case, by following the least commitment principle, the refinement operations tend to avoid adding to the token network those constraints which are not strictly needed. Starting from an initial node corresponding to a token network containing the problem requirement, the search aims at finding a final node containing a solution plan that correctly achieves the required goals. This algorithm is represented as a recursive non-deterministic schema in Figure 2.11.
procedure TP(\(\pi\))
\[
\text{flaws} \leftarrow \text{OpenGoals}(\pi) \cup \text{Threats}(\pi) \cup \text{Disjs}(\pi)
\]
if \(\text{flaws} = \emptyset\) then return \(\pi\)
select any flaw \(\phi \in \text{flaws}\)
\(\text{resolvers} \leftarrow \text{Resolve}(\phi, \pi)\)
if \(\text{resolvers} = \emptyset\) then return \(\bot\)
non-deterministically choose a resolver \(\rho \in \text{resolvers}\)
\(\pi' \leftarrow \text{Refine}(\rho, \pi)\)
return TP(\(\pi'\))

Figure 2.11: The Timeline-based Planning (TP) procedure.

The TP procedure is called initially with the token network \(\pi_0\) containing the problem requirement. Each successful recursion is a refinement of the current plan according to a given resolver. The correct implementation of the non-deterministic “choose” step is the following:

- When a recursive call on a refinement with the chosen resolver returns a failure, then another recursion is performed with a new resolver.
- When all the resolvers have been tried unsuccessfully, then a failure is returned from that recursion level back to a previous choice point. This is equivalent to an empty set of resolvers.

Before moving further, let us detail the variables and the procedures that are used in TP:

- \(\text{flaws}\) denotes the set of all flaws in \(\pi\) provided by procedures OpenGoals, Threats and Disjs; \(\phi\) is a particular flaw in this set.
- \(\text{resolvers}\) denotes the set of all possible ways to resolve the current flaw \(\phi\) in partial plan \(\pi\) and is given by the procedure Resolve. The resolver \(\rho\) is an element of this set.
- \(\pi'\) is a new plan obtained by refining \(\pi\) according to resolver \(\rho\) through the use of the Refine procedure.

\text{OpenGoals}(\pi). This procedure finds all the tokens in \(\pi\) which are not causally supported. Notice that this procedure can be efficiently implemented with an agenda data structure from which draw out flaws. For each new token \(t\) in \(\pi\), \(t\) is added to the agenda; the token is removed from the agenda if it is either unified or its associated rule is applied.

\text{Threats}(\pi). Threats are sets of possibly conflicting tokens like different values overlapping on a state-variable, resource overuses as well as resource overproductions and resource overconsumptions. Notice that, compared to [74], resource conflicts are included into the set of flaws.
2.3. PLANNING WITH TIMELINES

**Disjs (π).** These correspond to a syntactical facility that allows one to specify in a concise way several rules into a single one. Although they lead to an exponentially larger search space, they can be easily managed by including them into the same agenda data structure. Whenever the draw out flaw corresponds to a disjunction, a non-deterministic choice, among the available disjuncts, takes place. Disjunctions are also used for choosing the value of object variables.

**Resolve (φ, π).** This procedure finds all ways to solve a flaw φ. If φ is an open goal $g_i$ then its resolvers are either of the following:

- An unification constraint $g_i \equiv t_i$ if there is a token $t_i$ already in $π$, that is compatible with $g_i$ (i.e., the variables of $g_i$ and $t_i$ can be made pairwise equal).

- The application of the rule associated with the goal $g_i$. This resolver adds a new token $t_j$ for each target token $t_j \in R$, a new constraint $c_j$ for each constraint $c_j \in R$ and a new disjunction $d_j$ for each disjunction $d_j \in R$.

If φ is a threat among a set of tokens, its resolvers dependent on the nature of the timeline on which the threat arises and might include temporal constraints for ordering in time each couple of the conflicting tokens as well as constraints on their values reducing, for example, the amount of resource consumption or, even, constraints on their $\tau$ variables “moving” the tokens on other timelines. Finally, if φ is a disjunction, its resolvers are the disjuncts of the disjunction.

**Refine (ρ, π)** This procedure refines the partial plan $π$ with the elements in the resolver, adding to $π$ constraints, new tokens and new disjunctions. This procedure is straightforward: no testing needs to be done because we have checked, while finding a resolver, that the corresponding constraints are consistent with $π$. Refine just has to maintain incrementally the the agenda.

It is worth noting that while the choice of the resolver is a non-deterministic step (i.e., it may be required to backtrack on this choice), the selection of a flaw is a deterministic step (i.e., there is no reason to backtrack on this choice) as all flaws need to be solved before or later in order to reach a solution plan. Similarly to the partial-order case, however, the order in which flaws are processed is very important for the efficiency of the procedure yet is unimportant for its soundness and completeness. Moreover, a deterministic implementation of the TP procedure should rely on algorithms like A* or IDA* otherwise the search may keep exploring deeper and deeper a single path in the search space, adding indefinitely new tokens to the partial plan and never backtracking. As a consequence, similarly to the partial-order planning case, choosing the right flaw and the right resolver becomes a crucial aspect for coping with the computational complexity and hence efficiently generating solutions.

**Solving the rover domain.** In case of the rover domain there are, initially, two tokens in the token network representing the two facts while the goal is in the agenda. Additionally, the token network contains two constraints that force the two facts to start at time 0. The algorithm selects the initial TakingPic (1, 5, 3, 7) goal, let us call it
\( \phi_0 \), as the sole flaw returned by the OpenGoals (\( \pi \)) procedure. Since it is not already known how to achieve this goal, the Resolve (\( \phi_0, \pi \)) procedure returns the sole resolver, let us call it \( \rho_0 \), corresponding to the token's associated rule application. Finally, the Refine (\( \rho_0, \pi \)) procedure applies the rule and adds the token to the token network. As a consequence of the rule application, the two goals LookingAt (3,7) and At (1,5), let us call them, respectively, \( \phi_1 \) and \( \phi_2 \), are added to the agenda for further resolution. At the next step one of the flaws of the agenda is selected like, for example, \( \phi_1 \). Since there is no other token available assuming the same value, its corresponding rule is applied and the token is added to the token network. The application of the last rule introduces a disjunction flaw, let us call it \( \phi_3 \), into the agenda which now contains the previous \( \phi_2 \) and the new \( \phi_3 \) flaws. Again, a flaw from the agenda, for example \( \phi_2 \), is selected and, again, a rule is applied. A goal GoingTo (1,5), called \( \phi_4 \), is added and selected for resolution through its corresponding rule application which, finally, leads the introduction of a new goal At (?x,?y) which can be called \( \phi_5 \). It is worth noting that the Resolve (\( \phi, \pi \)) procedure, till now, has always returned a single resolver, hence, all the above steps were mandatory and did not require any decision to be taken. However the agenda, now, contains the two flaws \( \phi_3 \) and \( \phi_5 \) each resolvable through different ways. The \( \phi_3 \) flaw, indeed, represents the disjunction described into the LookingAt rule and can be solved either applying the first disjunct or the second. The \( \phi_5 \) flaw, on the other hand, can either represent the initial At (0,0) fact, in which case can be resolved through an unification, or not, in which case can be resolved through its corresponding rule application. The best choice is, for both of the flaws, the first option. In case of the \( \phi_5 \) flaw flaw, its corresponding token is unified with the initial fact resulting in the constraining of the variables to be pairwise equal (i.e., \( x = 0, y = 0, s = 0 \), etc.). In case of the \( \phi_3 \) flaw, the disjunct introduces a new RotatingTo goal whose rule application would introduce a LookingAt (?pan,?tilt) which, in turn, would unify with the initial fact resulting in a solution plan. Figure 2.12 represents a partial plan for the rover domains. Boxes represent tokens with their predicate names and their parameters. Solid arrows represent causal relations while dashed arrows represent the equality constraints introduced by the application of the rules. While the solid boxes represent the tokens in the token network, the dotted ones represent the unified ones. Finally, dashed boxes represent tokens which are still in the agenda. It is worth noting that, by unifying the dashed token in the Figure with the one on top of it, we obtain the solution described above.

### 2.3.3 Timeline-based planning vs other approaches

Rather than through propositions, timeline-based planning models the world by means of different features which evolve in time. Such features are represented by the temporal evolution of the timelines as a consequence of the tokens which are imposed on them. Before going further within the dissertation, it is worth highlighting some of the main differences in this way of making automatic planning compared to other approaches.
A different representation of the state. Timeline-based frameworks represent the states of the state transition system $\Sigma$ in a different manner compared to those of classical and temporal planners. Specifically, the proposition symbols $L$ are replaced by timelines which, in a more compact manner, can represent complex behaviors through
tokens. In this regard, it is worth to mention the SAS+ formalism (see [3]) which represents a planning problem using multi-valued state-variables instead of the propositional symbols \( L \). Timeline-based planning goes further in this direction representing values through predicate symbols and parameters. It is worth to note that, by allowing numeric variables in the token parameters, the size of the state space could potentially become infinite.

**Facts and goals.** Although there are some extension of classical planning (i.e., timed initial literals and trajectories) which try to mitigate this gap, most of the approaches at planning describe the state of the world by means of a collection of facts which are true at the beginning of the planning horizon while goals describe the characteristics that a desired state, eventually, must have. In other words, planning problems are usually described by means of the initial state, a set of possible actions and a set of goal states which, eventually, have to be reached by applying the actions. This approach precludes the possibility of introducing some desired behavior **within** the plan. Classical and temporal planning approaches address this lack through the introduction of state trajectories [55]. The unified representation of concepts through the tokens, together with a richer model of time, however, allow timeline-based planning framework to natively represent such desired behaviors.

**No distinction between actions and states.** Classical and temporal planning approaches are historically characterized by the dichotomy between agents and environment. In such approaches, the agents observe the environment and, consequently, perform some actions which, in turn, end up to alter the environment. It seems natural, therefore, to separate these two concepts. Timeline-based planning, on the contrary, uses the unified concept of token to represent both the state of the world and the actions of the agents. The motivations underlying this deviation mostly relies in the higher flexibility in imposing richer constraints among actions and the state of the world. This allows, for example, in contrast to classical planning (also in the extended version with time, i.e., PDDL2.1), to express natively the requirement that a proposition must hold for an interval of time. Furthermore, despite not having its own conceptual autonomy, agent’s actions have their own representation allowing us to impose constraints among actions as, for example, directly limiting time between two actions (e.g., to serve a dish, after it has been cooked, yet before it gets cold). Again, by combining temporal planning and timed initial literals, it is still possible, for temporal planning approaches, to achieve, through some artifacts, similar results. Modeling such problems into these formalisms, however, represents a challenge both for the planner and for the user who has to model the problem.

**Exogenous events.** Since there is a single primitive for representing information (i.e., the token), such primitive might be used for representing exogenous events, raising at execution time, as well. Timeline-based planning can hence more easily adapt to unforeseen events avoiding, in some cases, the need to regenerate plans from scratch in order to tackle the emergent states. As regarding this topic, it is worth to mention the PDDL+ formalism described in [44] which introduces the possibility to reason on
2.4. THE PERFORMANCE GAP

Like classical planning actions, PDDL+ events are modeled as instantaneous state transition functions which, however, can have numeric preconditions and effects. The key difference from classical planning actions, however, consists in the fact that PDDL+ events cannot be selected by the planner during the development of a plan but must be applied whenever their preconditions are verified. Therefore, although these types of events go in the direction of handling situations not directly controllable by the planner, they are still bound to the planning phase and not to the execution one.

Flexible plans. Timeline-based planning natively represents plans by means of a network of constraints. Similarly to partial-order planning, some of the involved variables might be maintained flexible. Usually, start and end times of tokens are left flexible allowing the final plan to represent *envelops* of plans. This flexibility assure *robustness* during plan execution, especially in such situations in which it is not possible to exactly know in advance the duration of some activities.

Modularity. Most of the modern artifacts are made of a number of separated components connected together. The ability to describe such components each other independently, proves to be a precious help in managing complexity at modeling time. Most of the Timeline-based planning frameworks allow decomposition of the domain model, resulting in a reduction the overall complexity at design phase.

2.4 The performance gap between timeline-based and other approaches

As shown in 2.1.1, the idea underlying classical planning heuristics consists in estimating the minimum distance $\Delta^*(s, g)$ from a state $s$ to a solution state containing all the propositions in $g$. This estimation is useful for taking decision during the search phase. The given hint is: choose the action that will lead to a state which is the closest as possible to a solution. How to estimate this distance depends on the chosen heuristic and, as already mentioned, represents, more in general, the main topic of a research field called *heuristic planning*. Intuitively, the more accurate the estimation is, the harder is to compute it, yet the more it will lead the search phase straight toward a solution, producing shorter plans and avoiding computationally expensive backtracking steps. Despite the introduction of concurrency, the approaches used to make efficient the reasoning in temporal planning follow a similar idea. Specifically, actions having a duration (a.k.a. durative actions) are treated as a couple of classical planning actions, one of which starts the action and the other concludes it. Different strategies are then applied to manage the concurrency [27, 25, 26].

Compared with classical and temporal planning, timeline-based planning has a clear advantage in being able to decompose problems into subproblems. Although these planners capture elements that are very relevant for applications, their theories are often quite challenging from a computational point of view and their performance is rather weak compared with those of other state of the art planners. As we have seen, timeline-based planners are mostly based on the notion of partial-order planning
and have almost neglected advantages in classical planning triggered from the use of Graphplan and/or modern heuristic search [9, 10]. Similarly to the partial-order approach, indeed, timeline-based planning suffers for not representing states directly, making it harder to estimate “how far” a partial plan is from being a solution. At present time, there is less understanding of how to compute accurate heuristics for timeline-based planning than for classical and temporal planning. As a consequence, timeline-based planners are, typically, inherently quite inefficient and rely on a careful engineering phase of the domain, possibly supported by the definition of domain-dependent heuristics. With a few exceptions (e.g., [7, 108]), the search control of these planners has always remained significantly under explored. 

In general, the different branching schema does not seem to allow effective pruning mechanisms during search hence dead-ends are discovered too late and, as a consequence, the size of the search tree is likely to explode. Similar to most partial-order planners, timeline-based planners tend to reason about the sole elements in the current partial plan [111], widely relying on the sole temporal reasoning aspects, and mostly overlooking possible subsequent plan refinements. The key question is: “how is it possible to reason on tokens that have not yet been added in the current partial plan, since some rules have not yet been applied?’’. When building and using the planning graph, Graphplan reasons, in a limited way, on all the possible plans underlying a domain specification, hence producing a wider view of the problem to be solved (the same holds for the heuristic search approaches based on [10] and derivatives). 

The order in which flaws are selected by the search algorithm and, particularly (since all of them must be solved), the choice of the resolver, have been recognized as key issues. Intuitively, solving, at first, the most complex flaws would allow an early detection of inconsistencies. An ideal flaw ordering would select, at first, a small strong backdoor, i.e., a set of flaws which, once solved, make the remaining problem easy to solve. Once chosen a flaw, however, it would be preferable to resolve it in a way that most likely would lead to a solution. The real challenge is then reduced at finding the “right” flaw and ordering its associated resolvers in an “right’’ way. Unfortunately, finding an optimal ordering is at least as difficult as solving the problem itself. Introducing some form of randomization into a given flaw and/or resolver ordering heuristic can cause great variability in performance. In conclusion, timeline-based planning leaves room for two types of heuristics: heuristics for selecting the next flaw, and heuristics for choosing the resolver of the selected flaw. The next chapters, indeed, will be focused on the effort in finding such heuristics.
Narrowing the Performance Gap with ILocco

As seen in Section 2.1, classical planning imposes strong restrictions in order to cope with computational complexity and performance issues. Possible enhancements come in two main flavors: either extend the classical planning state representation (by introducing durative-actions and numeric fluents [45], derived predicates and timed initial literals [41], state-trajectory constraints and preferences [55] and object-fluents\(^1\)) or follow the timeline-based way.

Exception made for some special cases in which concurrency is limited, namely those domains which are not temporally expressive [28], both alternatives lack of valid heuristics. This section presents ILocco, an initial attempt at bridging the performance gap between timeline-based planning and other more efficient approaches. ILocco represents the natural evolution of previous works on timeline-based planning carried on during the development of this thesis [29, 30, 31, 32].

3.1 The integrated Logic and Constraint Reasoner (ILocco)

Taking inspiration from both Constraint Programming (CP) and Logic Programming (LP), the ILocco (for integrated Logic and Constraint Reasoner) domain independent planning system is a timeline-based planner that allows to model both planning and scheduling problems according to a uniform schema. By pursuing the goal of going beyond the timeline-based representation, however, the ILocco framework slightly redefines the concepts of tokens and compatibilities as defined in Section 2.3.

The first consideration it is worth doing, in the case of existing formalisms for timeline-based planning, regards, indeed, the token concept. The idea underlying the token concept resides in the need for having a representation in time of a relation which is expressed by means of a predicate and its parameters. A token, indeed, is nothing more than a temporally scoped value applied to a timeline. Once introduced the concept of predicate and their parameters which, nevertheless, is required by the token

\(^1\)http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/kovacs-pddl-3.1-2011.pdf
formalism, it is possible to embed in it both the temporal variables and the object variable representing the timeline on which the token applies. The resulting data structure is just a first-order atomic formula called, for brevity, *atom*. More formally,

**Definition 19.** An atom is an expression of the form:

\[ n(x_0, \ldots, x_k) \]

where \( n \) is a predicate name, \( x_0, \ldots, x_k \) are constants, numeric variables or object variables.

The \( s \) and \( e \) temporal variables, representing the starting and the ending time of the token, as well as the \( \tau \) object variable, representing the timeline on which the token applies, *might* (as opposed to *must*, hence, the more generality) be included in the parameters of the predicate. The iLOC added value, in this case, lies in the possibility to reason even (and, if needed, at the same time) on those atoms which do not have such variables. It would not make much sense, as an example, to introduce a predicate such as \( \text{Know}(x) \), indicating the fact that an agent knows some concept \( x \), and to assign it to some temporal interval on a timeline.

Replacing tokens with atomic formulas has at least two benefits: (a) allows to unfasten the formalism from the timeline specific features without, of course, forbidding them, thus constituting an overwhelming set of concepts that can be represented and (b) avoids the introduction of an additional concept (i.e., the token), thus allowing us to rely more firmly on the theories related to first-order logic. By relying on the type of the \( \tau \) variable, if present among the atoms' parameters, it is possible to inject some domain-specific (and, thus, more efficient) knowledge within the framework without affecting the formalism. This is the case, for example, of the specific reasoning capabilities required in order to maintain the consistency of the state-variables as well as of the consumables and renewable resources.

A direct consequence of introducing the concept of atom is the requirement of a slight formal adaptation for the token network. Specifically, the token network concept is replaced by an analogous concept called *atom network*. Despite the names, the two concepts are, nonetheless, basically identical. More formally:

**Definition 20.** An atom network is a tuple \( \pi = (A, C) \), where:

- \( A = \{a_0, \ldots, a_k\} \) is a set of atoms.
- \( C \) is a set of constraints on the variables of the atoms in \( A \).

A further distinction with default timeline-based planners, implemented in iLOC, lies in the more general concept of type. In particular, the timeline types (i.e., state-variables, reusable and consumable resources, etc.) are special kinds of types. The advantage, here, is twofold: (i) the introduction of new types, as well as their domain-specific (and, thus, more efficient) knowledge, does not affect the formalism and (ii) it is possible to introduce new types, as well as their domain-specific knowledge, which are not necessarily timelines (i.e., do not involve explicit reasoning about time like, for example, in the case of a computational linguistic module).

Within iLOC, also the definition of rule slightly changes, becoming simpler and more expressive at the same time. Formally,
Definition 21. A rule is an expression of the form

\[ n(x_0, \ldots, x_k) \leftarrow r \]

where:

- \( n(x_0, \ldots, x_k) \) is the head of the rule, i.e., an expression in which \( n \) is a predicate name and \( x_0, \ldots, x_k \) are the parameters of the head (i.e., constants, numeric variables or symbolic variables).

- \( r \) is the body of the rule (or the requirement), i.e., either a slave (or target) token, a constraint among tokens (possibly including the \( x_0 \ldots x_k \) variables), a conjunction of requirements or a disjunction of requirements (in the latter case, the disjuncts might be characterized by a cost).

Finally, tLoC problems can be described as a set of objects (including, possibly, the timelines) each having associated a type, a set of rules and a requirement. More formally,

Definition 22. An tLoC problem is a triple \( P = (O, R, r) \), where:

- \( O \) is a set of objects.

- \( R \) is a set of rules.

- \( r \) is a requirement, i.e., either a fact atom, a goal atom, a constraint among atoms’ arguments, a conjunction of requirements or a disjunction of requirements (in the latter case, similar to rules, the disjuncts might be characterized by a cost).

The definition of rule, given above, does not differ much from the definition of a logic programming rule. While the head of a first-order rule can be replaced with the head of a timeline-based rule, the body of a first-order rule can be replaced with a requirement. Additionally, while first-order rules having the same head are considered disjunctive, timeline-based rules explicitly allow the definition of disjunctions. It is worth noticing that constraints are, ultimately, conditions which might be either satisfied or not and hence easily fit among the literals of the body of the rules. Consequently, the tLoC problem can be described by a collection of Horn clauses any of which belonging to three categories: (i) a Horn clause with exactly one positive literal (i.e., a definite clause, as described in Section 2.1) called a rule; (ii) a rule with no negative literals called a fact; and (iii) a rule without the positive literal called a goal. As a consequence, an tLoC problem strongly resembles a logic programming problem. This similarity will allow, in Chapter 7, to use the results presented in this thesis in contexts that go “beyond” the timelines, such as those related to logical programming and abductive reasoning. Unlike the logic programming case, however, rather than demonstrating goals, tLoC resolves flaws. This distinction is important because, except in

---

\*2This is, ultimately, nothing more than syntactic sugar which allows, in those cases in which disjunctive rules share some code, a more compact representation.
the case of basic flaws (i.e., facts, goals and disjunctions), flaws (and their resolvers) can be defined by the specific types from which they might arise (i.e., state-variables, reusable and consumable resources, etc.).

An additional consequence of the analogies with the logic programming approach lies in the possibility of sharing the resolution algorithms. From an operational point of view, indeed, ILOC uses an adaptation of the resolution principle [94] for first-order logic, extended for managing constraints in the more general scheme usually known as constraint logic programming (CLP) [2]. Starting from an initial atom network, containing the requirement as defined by the problem, the solver maintains an agenda of the current flaws and, incrementally, choses one of them. By exploiting the set of rules, the solver resolves the chosen flaw until the agenda becomes empty. The underlying constraint network, as usual, must always be maintained consistent. As already mentioned, the possibility for sharing the resolution algorithms can be seen as an added value also from the other side point of view. Specifically, the development of heuristics for solving timeline-based planning problems can be exploited to solve, also, constraint logic programming problems. To the best of the author’s knowledge such heuristics do not yet exist at all. Typical logic programming approaches, indeed, select goals according to a queue (i.e., according to a first in first out strategy) subject to how goals raise. The same approaches, furthermore, apply rules according to their order of declaration. Both these choices are rather poor from a performance perspective.

Figure 3.1 shows a general description of the ILOC reasoning engine. Specifically, the system maintains a set of rules (i.e., the domain theory). A working memory maintains information about the current objects and the current constraints among such objects (i.e., the atom network). An agenda maintains information about all the flaws to be resolved. Finally, an inference engine has the dual role of solving all the flaws of the agenda while maintaining consistent all the constraints among the objects. Notice that, despite the slightly different nomenclature, which allows greater flexibility in the representation of the domain models, the resolution algorithm is practically identical to a standard timeline-based resolution algorithm as described in Section 2.3. The final aim of the resolution process is, indeed, to remove all the flaws from the agenda while maintaining all the constraints consistent. The process follows a best-first search strategy, according to some heuristics which will be defined in the next sections of this chapter, proceeding until there are no more flaws into the agenda and while all the constraints in the working memory are consistent. Whenever the constraint network
3.2 Preliminary heuristics for ILOC

Since all the flaws must be solved sooner or later, there is almost no difference among which flaw is solved first. As already mentioned, however, selecting the “right” flaw heavily impacts with the efficiency of the resolution algorithm. A possible way for approaching the problem is to see the resolution procedure as a meta-CSP (refer to Section 2.2 for a brief introduction on CSPs). By considering the resolution process as a meta-CSP, indeed, it is possible to take inspiration from those heuristics, developed for solving CSP problems, which have proven to be effective. Specifically, decision points can be seen as CSP variables and decisions can be seen as choosing the assignment value for a given variable. Looking at things from this perspective might help to define the new heuristics. In particular, one of the most effective CSP strategies consists in choosing the most constrained CSP variables (in our case, flaws) first, so as to allow, in case, an early detection of inconsistencies, while trying to assign them the least constraining values (resolvers) first. Effectively measuring along these dimensions, however, might not be trivial. As a first attempt, [31] builds a data structure, called static causal graph (since it doesn’t change during the resolution process), aimed at producing some kind of information which might be used to guide the search process. Specifically, the causal graph has a node for each of the predicates that appear in the rules and, for every rule, an edge from each of the predicates that appear in the body of the rule to the predicate in the head of the rule. As an example, Figure 3.2 shows the static causal graph resulting from the rules of the rover domain as defined in Section 2.3.

Roughly speaking, this graph provides some initial information on how to achieve a specific goal because the slave atoms of the corresponding rule must also be in the atom network. The main underlying idea consists in creating a simple data structure (keeping Graphplan as a reference) that takes into consideration something more than the sole atoms in the current atom network. Such a data structure, indeed, allows to reason, although in a very limited way, about all possible plans, producing a wider view of the problem to be solved. By exploiting the topology of such a graph it is possible to extract the values for the above (variable and value selection) heuristics. Specifically, in choosing meta-variables, the furthest flaws from a solution will receive higher priority. On the contrary, meta-values (i.e., resolvers) will be assigned giving higher priority to those which are more likely to lead to a solution.
3.2.1 The AllReachable heuristic

The static causal graph roughly represents the causality of the domain from which it is extracted. In case of the rover domain, for example, in order for an At goal to be solved, a GoingTo atom must also be present in the atom network. In other words, for reaching an At goal, you have necessarily to pass for a GoingTo node. As a first attempt, the estimated cost for solving a goal, therefore, can be considered as the number of nodes from which the node relative to the predicate associated to the goal is reachable. Atoms having those predicates as their values, indeed, must also be present in the atom network. Such costs, in the case of the rover domain, are indicated by the numbers of Figure 3.2. As an example, the cost for solving an At goal is 1 (since At is reachable by the sole node GoingTo) while the cost for solving a TakingPic goal is 4 (since TakingPic is reachable by all the nodes GoingTo, At, LookingAt and PointingAt). This evaluation criteria provides an approximate estimation of the cost for solving a goal. Since the most constrained variable is preferable, higher priority is given to those flaws which have a higher estimated cost since they are, probably, harder to be solved. Suppose, as an example, the initial problem contains two goals At and TakingPic, the latter will be chosen, first, and a resolver will be applied so as to resolve it. Notice that this strategy is analogous to the strategy proposed in [108] for choosing a flaw since higher level flaws in a hierarchy are also, according to the AllReachable heuristic, the most expensive ones.

Additionally, by considering the cost for an atom network as the sum of all the flaws which still have to be solved, each, evaluated with the above criteria, it is possible to exploit the same graph also to estimate the cost for the resolvers, foreseeing, to some extent, the number of flaws which will have be solved before applying the resolver. This allows, additionally, to evaluate disjunctions as the minimum cost of its disjuncts. Suppose, as an example, two goals on the rover domain ask (a) to go at a location \( l_0 \) and (b) to either go at some other location \( l_1 \) or to take a picture on another location \( l_2 \). Notice that the latter can be achieved by means of a disjunction involving an At goal, on a first disjunct, and a TakingPic goal, on a second disjunct. Since both flaws are equally evaluated (i.e., 1), any of the two is chosen, e.g., the latter. Now, since the first disjunct would lead to a partial plan having cost 2, as the result of the contribution of the At goal (i.e., 1) plus the contribution of the first disjunct (i.e., 1), and the second disjunct would lead to a partial plan having cost 5, as the result of the contribution of the At goal (i.e., 1) plus the contribution of the second disjunct (i.e., 4), the former is chosen.

To sum up, when choosing the next flaw to be solved, choose the flaw having the maximum estimated cost, when choosing the next resolver to apply, choose the one which would lead to having the minimum estimated distance from the solution. [31] calls such a strategy AllReachable. The idea behind this strategy is to evaluate flaws by considering a kind of worst case scenario where none of the atoms unify. Clearly, this graph and, consequently, the costs for each of its nodes, solely depends on the rules and, thus, can be built once and for ever at the beginning of the solving process (whence the name static), allowing constant-time cost retrieval. The heuristic, however, completely neglects all the possible disjunctions (resulting from rules having the same head) of the domain theory. Furthermore, the arguments of the predicates are
3.3. EXPERIMENTAL EVALUATION

Figure 3.3: The AND/OR static causal graph associated to the rules of the rover domain and the costs estimated by the **MinReach** heuristic.

ignored as well as the constraints among them.

3.2.2 The **MinReach** heuristic

A slight improvement to the above heuristic can be achieved by introducing disjunctions into the static causal graph. Specifically, it is possible to enhance the causal graph by means of two special nodes representing conjunctions (AND nodes) and disjunctions (OR nodes). Figure 3.3 shows the enhanced static causal graph generated from the rules of the rover domain. The cost for solving a flaw is now evaluated as the minimum number, without repetitions, of all the nodes from which the node associated to the goal predicate is reachable. [32] calls such a strategy **MinReach** heuristic. As an example, the cost for solving a *LookingAt* goal is now reduced from 4 to 1 since the *LookingAt* node is reachable either from the *TakingPic* node, having cost 3 or from the *RotatingTo* node, having cost 1. Introducing the sole *RotatingTo* atom, however, is probably preferable than introducing a *TakingPic* atom which would require, also, an *At* atom and a *GoingTo* atom and, clearly, and far more preferable than introducing all of the above atoms as expected by heuristic **AllReachable**.

3.3 Experimental evaluation

To assess the value of our heuristics, we have endowed iLOC with the above described **MinReach** (MR) and **AllReachable** (AR) heuristics and tried to compare the resulting system with different planners on different benchmarking problems. Specifically, we have selected three planners that are interesting for their features and compared them with iLOC: VHPOP [100] shares with our planner the partial-ordering approach, OPTIC [6] and COLIN (see [24]) are both based on a classic FF-style forward chaining search [65]. All the tests have been executed with default configurations for every planner. It is worth to say that, although VHPOP is slightly dated, both OPTIC and COLIN are quite recent works.
The blocks world domain

We start the comparison by solving the Blocks World domain, a workhorse for the planning community. The reason for choosing this domain relies, basically, in its simplicity even if it is not that easy from a computational point of view. In this domain, the aspects related to temporal reasoning are practically absent. The actions, indeed, do not have a duration or, in any case, it can be considered negligible for the sake of solving the problem. These features make it a challenging problem for timeline-based planners that work well in those cases in which temporal aspects are predominant, yet less well when the logical/causal reasoning aspects prevail.

As known, in this domain a set of cubes (blocks) are initially placed on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. We used the 4-operator (i.e., pick-up, for picking a block from the table, put-down, for putting a block on the table, stack, for stacking a block on another and unstack, for picking a block from another block) version of the classic Blocks World domain, as found on the IPC-2011 website, as a starting point. Furthermore, we use a simpler variant of the general problem (usually called Tower) in which, in the initial state, all the blocks are on the table and whose goal is to stack all the blocks on a single tower. Specifically, for each block, we defined a state-variable for representing what is on top of the block (i.e., either another block or the value “Clear”) and a state-variable for representing if the block is on the table or not. An additional state-variable has been defined for modeling the robotic arm supporting values that represent either the arm holding a block or the value “Empty”. Finally, we defined an “Agent” complex type for modeling the agents’ actions. Rules have been defined so as to have an atomic formula for each effect of the PDDL actions as head and an atomic formula for the actions as body (modeled as a subgoal), aside from rules having an atomic formula for each PDDL action as head and an atomic formula for their preconditions (modeled as subgoals) and effects (modeled as facts) as body. Temporal constraints have been conveniently added for guaranteeing that preconditions precede actions and effects follow actions.

The Temporal Machine Shop domain

The Temporal Machine Shop problem is the only temporally expressive problem of the International Planning Competition (IPC) and, within the same competition, it is solved by the sole ITSAT planner (see [91]). The problem models a baking ceramic domain in which ceramics can be baked while a kiln is firing. Different ceramic types require a different baking time. While a kiln can fire for at most 20 minutes at a time (and then it must be made ready again), baking a ceramic takes, in general, less time, therefore we can save costs by baking them altogether. Additionally, similar to [91], we have slightly complicated the domain by considering the possibility for ceramics to be assembled, so as to produce different structures which should be baked again to obtain the final product. Specifically, for each kiln we defined a state-variable for distinguishing either the kiln is “Ready” or “on Fire”. In addition, each kiln has
associated a reusable resource for representing its capacity. For each ceramic piece we defined a state-variable for representing either the piece is “Baking” (with an additional parameter for representing the kiln in which is baking), or the piece is “Baked”, or the piece is “Treating”, or the piece is “Treated”. Similarly, for each ceramic structure we defined a state-variable for representing either the structure is “Assembling”, or the structure is “Assembled”, or the structure is “Baking” (with an additional parameter for representing the kiln in which it is baking), or the structure is “Baked”. Rules force these values to appear in time, in each state-variable, in the intuitive manner (i.e., in the order in which these values have just been introduced). The interesting aspect, however, is that ceramic structures can bake concurrently with ceramic pieces both while (hence the temporal expressiveness) the kiln is firing.

The Cooking-carbonara domain

The Cooking Carbonara domain represents another temporally expressive problem in which the aim is the preparation of a meal, as well as its consumption by respecting constraints of warmth. Problems cooking-carbonara-\(n\) allow to plan the preparation of \(n\) dishes of pasta. The concurrency of actions is required to obtain the goal because it is necessary that the electrical plates work in a way that water and oil are hot enough to cook pasta and bacon cubes. It is also necessary to perform this baking in parallel to serve a dish that is still hot during its consumption. Specifically, for each plate we defined a reusable resource for representing its (unary) capacity. For each pot we defined a state-variable for distinguishing either the pot is “Boiling” (with an additional parameter for representing the plate on which is boiling) or the pot is “Hot”. For each pan we defined a state-variable for distinguishing either the pan is “Boiling” (with an additional parameter for representing the plate on which is boiling) or the pan is “Hot”. Each portion of spaghetti has associated a state-variable for distinguishing either the portion is “Cooking” (with an additional parameter for representing the pot in which is cooking) or the portion has been “Cooked”. For each bacon portion we defined a state-variable for distinguishing either the bacon is “Cooking” (with an additional parameter for representing the pan in which is cooking) or the bacon has been “Cooked”. Each egg has associated a state-variable for distinguishing either the egg is “Being beaten” or the egg has been “Beaten”. Finally, for each carbonara portion we defined a state-variable for distinguishing either the portion is “Cooking” (with an additional parameter for representing the plate on which should be cooked), or the portion has been “Cooked”, or someone is “Eating” the portion or the portion has been “Eaten”. Again, rules force values to appear in time, in each state-variable, in the intuitive manner (i.e., in the order in which these values have just been introduced). Furthermore, carbonara portions should be cooking after spaghetti, bacon and eggs have been correctly prepared, hence requiring spaghetti to be “Cooking” while the water in pots is “Hot” as well as bacon to be “Cooking” while the oil in pans is “Hot”. Finally, cooking carbonara portions, boiling water in pots and oil in pans should be performed while plates are available.
3.3.1 Results

Starting from the blocks world (the problem in which logical/causal reasoning prevails), we can see, in Figure 3.4, that despite the introduction of these heuristics, planners endowed with “classical heuristics” still perform significantly better than our approach. Thanks to the MINREACH heuristic, however, we were able to boost the system performance appreciably, allowing us to find solutions up to, approximately, one third of the time it was required before. Such partial results for iLOC should not come as a surprise since its current heuristics, as will be clearer in the next chapters, neglect much of the precious information like the initial working memory, the constraints among objects and the constraints defined within rules. Not considering such information is, probably, the explanation for the performance gap still existing between iLOC and COLIN or OPTIC – both planners coming from a longer history about causal reasoning.

Experimental results on the other domains (Figures 3.5 and 3.6) show that iLOC performs competitively with respect to COLIN and OPTIC. It is worth observing that the heuristics neither guarantee a substantial improvement nor their overhead produces a significant worsening (performance remains almost unchanged). This is explained by the fact that the temporal/resource reasoning features of iLOC are not much affected by the causal heuristics. Furthermore, even though COLIN performs better than iLOC, it is not able to solve problems with more than 50 ceramics since it runs out of memory (we used the default configuration for the planner). This aspect is much more evident in the Cooking Carbonara domain which, since it does not contain a maximum duration for plate firing, can be easily reduced to a basic scheduling problem and, as such, easier for the timeline formalism.

A separate discussion it is worth doing concerns the modeling capabilities of iLOC. All the competing planners use the PDDL2.1 language (see [45]) for modeling their planning problems and, in general, it is quite cumbersome to impose temporal constraints among plain PDDL actions. In the Cooking Carbonara domain, for example, it is important that the cooking happens before the eating but eating should not start too late to avoid that food becomes cold. In [77] a PDDL extension is proposed to overcome this issue and to model properly the domain, however, none of the available plan-
3.3. EXPERIMENTAL EVALUATION

Figure 3.5: Temporal machine shop.

Figure 3.6: Cooking carbonara.
ners supports this extension and thus they have been evaluated in a simplified domain in which the warmth constraint decays and dishes can be served anytime after they have been cooked. It is worth noting that this extension does not affect the expressiveness of the language. These constraints can be represented indirectly by forcing, through causal constraints, the application of actions. This trick, however, makes harder the modeling of the problem, by the modeler, as well as harder the resolution, by the planner, which can not exploit some knowledge of the domain since it is provided in an indirect way. This constraint is naturally captured in the ILOC modeling language by creating a rule having as head an action and as body a second action in conjunction with a constraint among the temporal parameters of the two actions.

3.4 The gap does not narrow enough

Finding the right balance between the informativeness and the computational complexity of a heuristic is not an easy task. In any case the current level of informativeness is not satisfactory and should be clearly increased. Although in a limited way, Graphplan and other planners based on heuristic search allow to reason, at once, about all possible plans (all the plans achievable unfolding the domain theory). The ALLREACHABLE and the MINREACH heuristics have been developed trying to reproduce these capabilities while keeping the original architecture. It is worth noting, indeed, that each of the nodes of the search space has its own atom network with associated its own flaw agenda. These data structures are dynamically modified during the resolution process. By solving a flaw in the agenda, indeed, requires removing it from the agenda and, as a consequence of applying a resolver, adding further flaws to the agenda as well as atoms and constraints to the atom network. The representation of such possible plans within the above graphs, however, is still too weak, leading to too uninformed search strategies. The above heuristics completely ignore the constraints within the rules’ bodies, solely relying on the pure causality relations among predicates. Similarly, predicate arguments and constraints among them, coming from the facts and goals initially appearing in the working memory, are neglected as well. All this information is directly available from the domain theory and from the initial problem specification and should be, somehow, exploited.

A first refinement step might be to consider the initial state of the atom network. Consider again the graph in Figure 3.3. Now suppose two At and LookingAt facts are initially present in the working memory. We might imagine that eligible subgoals, coming from the application of the rules, will eventually unify with the above facts. We might consider such information for enhancing the informativeness of the heuristic by pruning, in general, the causal graph. In such a circumstance, indeed, the cost for solving a TakingPic goal is now further reduced from 3 to 0. Indeed, despite they are two, we might consider the At and LookingAt subgoals as preferable, since they will be unified, compared to the sole RotatingTo subgoal which would be chosen by the MINREACH heuristic. Consider, furthermore, the graph in Figure 3.3. The two edges outgoing from node LookingAt represent the eventual unification of two atoms having a LookingAt predicate. Now suppose, as an example, that the initial state contains a fact stating that the pan-tilt unit is LookingAt (0,0) and a goal stat-
3.4. **THE GAP DOES NOT NARROW ENOUGH**

As `TakingPic(1, 5, 2, 7)`. Clearly, the `LookingAt(2, 7)` subgoal, generated from the `TakingPic` goal, will not unify with the `LookingAt(0, 0)` fact, hence, the choice for the disjunction goes back to be the `RotatingTo` one.

If we want to consider more information within our heuristic, a possible path to follow might be to replace, in the causal graph, predicates, directly, with atoms. This generally means considering, at the beginning of the planning process, all the atoms that will be part of all the possible plans, having the cautiousness of activating those that will be part of the solution. Unfortunately, this transition is not as straightforward as it might seem. The atom network concept, indeed, should melt with the static causal graph and, as a consequence, the flaw agenda should be highly revisited, representing no longer the flaws of the atom network which require to be solved. Additionally, atoms should be endowed with an additional variable for representing their state (i.e., `ACTIVE`, `INACTIVE` and `UNIFIED`) since not all of them, present in the atom network, will be part of the solution plan. Furthermore, an ad hoc data structure which does not propagate constraints in case of disjunctions (we recall that such disjunctions represent branches in the search tree), yet powerful enough to propagate constraints that allow to recognize ineligible unifications, should be created and integrated with the above graph. While the following chapter introduces such a data structure, Chapter 5 exploits the data structure and, referring to some of the concepts introduced in this chapter and on Chapter 2.1, introduces a new heuristic which is able to overcome the above limitations.
CHAPTER 3. NARROWING THE PERFORMANCE GAP WITH ILOC
The SMT-based Constraint Network

The tLoC system is largely inspired by OMPS [48] and, with it, shares some strengths and defects. As an example, while tLoC manages the temporal constraints by means of a temporal constraint network [36], other kinds of constraints, like those among non-temporal parameters, are managed through the Choco’s [90] constraint network. Although Choco can generate explanations for inconsistencies and, although it is possible to generate explanations for inconsistent temporal constraint networks [103], none of these is exploited for enhancing the search performance through backjumping. Furthermore, the Choco solver cannot take advantage of temporal inconsistencies nor the temporal constraint network can exploit Choco’s constraint network inconsistencies. Additionally, since no information is learned from the inconsistencies, it is not uncommon that the overall solving procedure reaches different situations that share the same issue, resulting in a marked reduction in performance. Among the most interesting aspects of tLoC, the introduction of the concept of type is aimed at encapsulating the reasoning power of dedicated solvers and/or the complexity of non-symbolic state maintenance. Furthermore, tLoC introduces the concept of atom network, i.e., a constraint network of atoms on the different components which the planner leverages to propagate new decisions and constraints that are synthesized during planning.

Although this multiple solvers architecture is, in principle, commendable, its implementation presents some weaknesses which might make it inefficient. Specifically, the atom network is the only available tool which allows communication among such different dedicated solvers. As a consequence, exchanged information results to be too abstract, preventing the solvers to help each other when problems arise. The only exchanged information, indeed, is YES (i.e., no inconsistency has been found) and NO (i.e., some inconsistency has been found). In case of inconsistencies it is not possible, for a given solver, to generate any failure explanation and, even worse, tLoC lacks of a common language which might be used to exchange information among solvers.

By relying on the concept of reification [35, 75], a first attempt to narrow this performance leak has been made in [29]. Specifically, reified constraints are a typical CSP technique which make it possible to express constraints involving logical connectives such as disjunctions, implications, and negations. A reified constraint is obtained, sim-
Chapter 4. The SMT-Based Constraint Network

ply, associating a propositional variable to a constraint. The role of such a variable is twofold: control the application of the constraint (i.e., in case the variable is asserted, the constraint is enforced, in case the variable is negated, the negation of the constraint is enforced) and verify the satisfiability of the constraint (i.e., in case the constraint is satisfied, the variable is asserted, in case the constraint is unsatisfied, the variable is negated). To further clarify this concept, let us suppose a constraint network having two object variables \( x \in \{a, b\} \) and \( y \in \{a\} \). In order to reify a \( x = y \) constraint we can associate to it a propositional variable \( p \equiv \| x = y \| \). The \( p \) variable is, initially, unassigned. While asserting \( p \) would result in the removal of the \( b \) value from the \( x \)'s domain (or, analogously, in asserting the \( x \neq y \) constraint). Suppose, for some reason, the \( b \) value is removed from the \( x \)'s domain, the \( x = y \) constraint is necessarily satisfied, hence, the TRUE value is assigned to the \( p \) variable. Finally, in case the \( a \) value is removed from the \( x \)'s domain, the \( x = y \) constraint is unsatisfiable, hence, the FALSE value is assigned to the \( p \) variable. The overall idea, pursued in [29], was to use the propositional logic language as a glue among the different dedicated solvers and to apply the reification concept also to the resolvers, so as to allow a uniform language for conflict analysis results and, thus, enhancing performance by exploiting no-good learning and non-chronological backtracking.

Also in this case, however, a commendable idea was realized in the wrong way, resulting in other kinds of inefficiency issues. The main issue, in this case, derived from the use of arc-consistency techniques (see, for example, [35, 75]) for propagating numeric constraints which, in the case of inconsistent instances and large domains, might be prohibitively expensive. Think of, for example, the trivial case in which two integer variables \( x \) and \( y \) are constrained by \( [x < y] \land [x > y] \). Such a problem is clearly unsatisfiable. Suppose, however, the initial domain of the variables is \( x \in [0, +\text{inf}] \) and \( y \in [0, +\text{inf}] \), arc-consistency starts to perform bound propagation resulting in \( x \in [1, +\text{inf}] \), \( y \in [2, +\text{inf}] \), \( x \in [3, +\text{inf}] \), etc. proceeding, clearly, ad infinitum. Although some attempts were made to narrow this inefficiency, they resulted to be insufficient and the solvers, consequently, inefficient. As demonstrated in the previous minimal example, arc-consistency on numeric variables is applicable only on (small) bounded variables. Choosing such bounds, however, might not be straightforward since there is a risk of losing completeness due to overly stringent domains. Although most of the CSP solvers (e.g., Choco, Gecode [51], JaCoP [70], etc.) provide convenient software interfaces to exploit their underlying constraint networks, all of them, to the best of the author’s knowledge, exploit arc-consistency techniques for propagating numeric constraints and, hence, result to be not suitable for our needs.

The idea of using propositional logic as a common language to exchange information among different solvers appears to be the leading concept of Satisfiability Modulo Theory (SMT) [98], a technique which has already been successfully used for solving planning problems in, for example, SMTPlan [13]. Specifically, by leveraging on the recent improvements of SAT solvers (e.g., [83, 42]), SMT solvers (e.g., [33], [21], [23]) use the concept of theory in a similar and, in some respects, more general way than the OMPS components. Furthermore, thanks to the work described in [37], it is possible to replace arc-consistency in managing numerical constraints and, relying on the simplex method, efficiently reason on linear arithmetic while leaving numeric
variables’ domains unbounded. Once again, however, the particular structure of the addressed problem is such that the existing techniques are not sufficient to efficiently solve it. The reason is twofold. Firstly, SMT based heuristics are too weak for supporting the search of a constraint-based planning problem, resulting in the exploration of unprofitable sections of the search space. In other words, we need to build higher level heuristics allowing us to take advantage of the intrinsic structures of the faced planning problem. Secondly, in building such a heuristic, we need to create an extremely huge SMT problem containing many variables whose values, mostly, we are not interested at all. This is the case, for example, of the variables associated to those atoms that are not part of the solution since they are neither active nor unified. There is no reason to waste computational power in reasoning about them, yet the problem structure is useful for extracting information that can be exploited to guide the search process. Unfortunately, although most of them are open-source, common SMT solvers, conversely to CSP solvers, are more closed, with less or none documentation for defining new heuristics nor for redefining search strategies. Hence the need for building something from scratch.

The following sections present an SMT-based constraint network leveraging on a propositional constraint network which controls the associated theories. Two theories are also implemented, i.e., a linear real arithmetic theory, for managing numeric variables and linear constraints among them, and a var theory, for managing object variables and (in)equality constraints among them. A new timeline-based solver called ORATIO, a theory itself, will be built on top of such network exploiting it both for producing a new heuristic, based on a relaxed version of the timeline-based planning problem, and for solving the whole timeline-based planning problem.

### 4.1 The SAT core

At the core of the needed data structure there is a component which we have named *sat_core*. It is, basically, a MiniSAT [42] implementation stripped of those characteristics strictly related to the search like, for example, the DPLL algorithm and the variable/value selection heuristics. The implementation retains, however, those characteristics (i.e., the *watched literals*) aimed at efficiently propagating the propositional constraints which constitute the hallmark of Chaff [83]. At the same time, the idea of generating an explanation as a result of a conflict analysis, as presented in [99], is preserved, allowing *backjumping* (a.k.a. *non-chronological backtracking*). It comes out a backtrackable data structure, which might be called *propositional constraint network*, whose characteristics are: (i) efficient propagation of propositional constraints, (ii) the ability to generate explanations (and, consequently, no-goods) in case of conflicts and (iii) transparent integration with theories. Most of the following code is an adaptation of MiniSAT.

The external interface of such a data structure, through which a user application can specify *sat_core* problems, is the following.

```cpp
class sat_core {
    var new_var()
    bool new_clause(vector<l it > lits)
}
```
CHAPTER 4. THE SMT-BASED CONSTRAINT NETWORK

```cpp
bool assume(lit p)
bool check()
bool check(vector<lit> lits)
lbool value(var x)
}
```

Intuitively, variables are introduced by calling `new_var()`. From these variables, clauses are built and added by means of the `new_clause()` procedure which, in some cases, might detect trivial inconsistencies (in which case it returns `false`). The `assume()` method is responsible for assuming a literal after creating a branching point. Furthermore, `check()` propagates clauses, returning `false` if the problem is `unsatisfiable` and `true` if no inconsistencies have been found (which, in general, does not necessarily implies that the problem is solved!). Similarly, the second `check()` method first assumes the given literals and then propagates clauses, returning `false` if the problem is, assumed the given literals, `unsatisfiable` and `true` if no inconsistencies have been found. Finally, `value()` allows for retrieving the value (`true`, `false` or `unassigned`) of the given propositional variable in the current state (or, in case of a conflicting state, an unreliable value).

The `lit` abstract data type, representing literals, is simply described by a variable and the sign of the variable within the literal. Utility methods can be used for returning the negation of a literal (i.e., a literal having the same variable but different sign) and the value of a literal (exploiting the above mentioned `value()` method for retrieving the value of a variable). Given the simplicity of the data structure, further details are not provided.

```cpp
class lit {
    lit op=();
    var v;
    bool sign;
}
```

`lbool` is used for representing lifted propositional domains, hence containing the elements `true`, `false` and `unassigned`:

```cpp
enum lbool { true, false, unassigned }
```

Finally, `var` is a type synonym for an `unsigned int`, with two special constants ⊥var = 0 and ⊤var = 1 for representing, respectively, the `false` and the `true` propositional constants.

### 4.1.1 The solver state

Since the `sat_core` must never be in a conflicting state, there is not anything that remembers it, hence, any method that puts the network in a conflicting state must communicate it. A number of things, however, need to be stored for representing the state:

```cpp
vector<clause> consrs // the list of all problem constraints..
// for each literal 'p', a list of constraints watching 'p'..
vector<vector<clause>> watches
queue<lit> prop_q // the propagation queue..
vector<lbool> assigns // the current assignments..
// the list of assignments in chronological order..
```
4.1. THE SAT CORE

From a quick comparison with the MINISAT implementation it is immediately evident the removal of those components responsible for the heuristics like, for example, the activity of variables for the variable selection heuristic. Furthermore, the current implementation does not keep track of the learnt clauses which, unlike in MINISAT, are never removed. This choice has been made as an implementation simplification. Future versions, indeed, could take account of learnt clauses in order to implement, for example, random restart strategies. In addition to the MINISAT implementation, however, a reference to all the bound theories is maintained and, moreover, for each variable, a collection of theories interested in the variable assignment is maintained.

Together with the above state variables, the following helper methods are also defined:

```cpp
class sat_core {
  bool value(var x) { return assigns[x]; }
  bool value(lit p) { return p.v ? assigns[x] : !assigns[x]; }
  unsigned index(lit p) { return p.sign ? p.v << 1 : (p.v << 1) ^ 1; }
  bool root_level() { return trail_lim.empty(); }
  void bind(var x, theory th) { binds[x].add(th); }
}
```

Notice that multiplications by 2 are replaced with shift (and bitwise) operators. Since these operators do not involve arithmetical operations are more efficient on most of the available architectures. In simple terms, the index of a literal is given by the index of its variable times two, if the literal is positive (e.g., \( x \)). On the other hand, the index of a literal is given by the index of its variable times two plus one, if the literal is negative (e.g., \( \neg x \)).

The propagation of the sat_core constraints is largely inspired to that of MINISAT which, in turns, is an adaptation of CHAFF [83]. Specifically, for each literal, a list of constraints is kept. These are the constraints which may propagate unit information whenever the literal becomes FALSE\(^1\). Note that there is no need for propagating information whenever a literal becomes TRUE since, a clause which is known to be satisfied does not propagate information. As a consequence, two unbound literals \( p \) and \( q \) are selected and references to the clause are added to the lists of \( \neg p \) and \( \neg q \) respectively. The literals are said to be watched and the lists of constraints are referred as to be the watcher lists. Whenever a watched literal becomes FALSE, the constraints in its associated watched list are checked to see if information might be propagated. Notice

\(^1\)The semantic, here, is the inverse of the one used in MINISAT and in CHAFF which propagate unit information whenever the literal becomes TRUE. There is no reason for choosing a semantic respect to the other except for the fact that our semantic seems more intuitive and easier to understand. Switching to the MINISAT semantic simply requires inverting the \( \neg \) of the index() function.
that, when backtracking, no adjustment to the watcher lists need to be done, therefore, backtracking is very cheap. It is worth to notice that, conversely to a standard arc-consistency algorithm, the watch list is based on literals and not on variables, making the propagation algorithm more selective. Furthermore, each clause watches only on two literals rather than on all of its literals. Although this strategy makes the propagation of a clause slightly more complicated, the overall number of propagation is largely decreased. The combination of these two choices, i.e., watching literals rather than variables as well as watching only two literals rather than all the literals of each clause, makes the CHAFF implementation extremely efficient.

4.1.2 Variables and constraints

The first code fragment that will be presented is the one that allows the creation of new propositional variables:

```cpp
var new_var() {
    var x = assigns.size()
    watches.add(vector<clause>())
    watches.add(vector<clause>())
    assigns.add(Unassigned)
    level.add(0)
    reason.add(null)
    binds.add(vector<theory>())
    return x
}
```

Intuitively, the new variable is given by the size of the assigns vector. The procedure makes room for the watcher lists associated both to the directed and to the negated literal and the Unassigned value is assigned to the new variable. Other parameters are also set, the meaning of which will be clearer soon.

Although MINISAT can, in principle, handle arbitrary constraints over propositional variables, we have limited ourselves to the implementation of the sole clause constraint. Its propagate() method is called in case the clause is found in the watcher list during propagation of unit information \( p \). In this case the constraint is removed from the list and is required to add itself into a new (or into the same) watcher list. If successful, true is returned; conversely, if a conflict is detected, false is returned. In the latter case the clause represents the cause of the inconsistency and, as will be shown later, will be used to generate an explanation of the conflict. Since there is no need for any SAT related heuristics code, the implementation of the clause constraint becomes straightforward being limited to its constructor and to the propagate() method which, as suggested by [42], should be the primary target for efficiency.

```cpp
class clause {

    clause(sat_core s, vector<lit> lits) : s(s), lits(lits) {
        s.watches[s.index(~lits[0])].add(this)
        s.watches[s.index(~lits[1])].add(this)
    }

    bool propagate(sat_core s, lit p) {
        // make sure false literal is lits[1]..
    }
}
```
### 4.1. THE SAT CORE

Implementing the `add_clause()` method of the `sat_core` API is just a matter of creating a new clause instance and adding it to the `constrs` vector of stored constraints. Some optimizations, however, can be implemented. Clauses which are already satisfied (i.e., at least one of their literals is `True`) are not created at all. Similarly, clauses which represent tautologies (i.e., both `p` and `¬p` occur in their literals) are not created at all. Additionally, `False` literals can be filtered out before creating a new clause. In case the resulting list of literals becomes empty (i.e., all of the initial literals were `False`) a trivial conflict is detected and `false` is returned. Finally, in case the resulting list contains a single literal, the clause is unit and the single literal representing the unit fact is enqueued for propagation without any need of creating a new clause. Notice that the current implementation allows calls of the `add_clause()` method only at top-level or there might be problems in the filtering of literals already assigned at lower decision levels.

```cpp
bool add_clause(vector<lit> lits) {
    vector<lit> ls
    for (lit l : lits) {
        switch(value(l)) {
            case True: return true // the clause is already satisfied..
            case Undefined:
                if (ls.contains(¬l)) return true // the clause is a tautology..
                else if (!ls.contains(l)) ls.add(l)
            }
        }
        if (ls.empty()) {
            return false // inconsistency..
        } else if (ls.size == 1) {
            enqueue(ls[0]) // unit fact is enqueued (cannot fail)..
        }
    }

    // if 0th watch is true, the clause is already satisfied..
    if (s.value(lits[0]) == True) {
        s.watches[s.index(p)].add(this)
        return true
    }
    // we look for a new literal to watch..
    for (unsigned i = 1; i < lits.size(); i++) {
        if (s.value(lits[i]) != False) {
            lit tmp = lits[1], lits[1] = lits[i], lits[i] = tmp
            s.watches[s.index(¬lits[1])].add(this)
            return true
        }
    }
    // clause is unit under assignment..
    s.watches[s.index(p)].add(this)
    return s.enqueue(lits[0], this)
}
```
4.1.3 Reified constraints

As already mentioned, reifying a constraint means associating a propositional variable to the constraint so as to represent whether the constraint is satisfied or not. In other words, reifying a constraint means that we allow constraints not to be satisfied. Since reified constraints might be useful, the sat_core provides a software interface for defining them. Specifically, the following code defines a reified conjunction:

```java
var new_conj(vector<lit> lits) {
    var c = new_var();
    vector<lit> tmp
    tmp.add(lit(c, true))
    for (lit l : lits) {
        new_clause({ l, lit(c, false))
        tmp.add(-l)
    }
    new_clause(tmp)
    return c
}
```

The above method introduces new clauses, in the number of the literals of the conjunction plus one, and a new variable, which will be returned, whose value determines whether the constraint is satisfied or not. In other words, while assigning TRUE to the returned variable forces the conjunction to be satisfied, assigning FALSE to it forces the conjunction to be not satisfied. Similarly, in case the constraint is necessarily satisfied, TRUE is assigned to the returned variable and, in case the constraint becomes unsatisfiable, FALSE is assigned to the returned variable.

Similarly to the conjunction case, the following code defines a reified disjunction:

```java
var new_disj(vector<lit> lits) {
    var c = new_var();
    vector<lit> tmp
    tmp.add(lit(c, false))
    for (lit l : lits) {
        new_clause({ -l, lit(c, true))
        tmp.add(l)
    }
    new_clause(tmp)
    return c
}
```

Finally, the sat_core interface provides the possibility for defining an “exactly one” constraint stating that exactly one, from a collection of literals must be TRUE. Although there exist more efficient implementations as, for example, the one proposed in [71], a naive encoding, combining the at-most-one and the at-least-one constraint, requiring the introduction of \(O(n^2)\) new clauses, is used:
4.1. THE SAT CORE

```cpp
var new_exct_one(vector<lit> lits) {
    var c = new_var()
    vector<lit> tmp
    tmp.add(lit(c, false))
    for (unsigned i = 0; i < lits.size(); i++) {
        for (unsigned j = i + 1; j < lits.size(); j++) {
            // the at-most-one constraints
            newClause( {¬lits[i], ¬lits[j], lit(c, false)} )
        }
    }
    tmp.add(lits[i])
    // the at-least-one constraint
    newClause(tmp)
    return c
}
```

Similarly to the conjunction case, the returned variables of the disjunction and of the exactly one cases determine whether the constraints are satisfied or not. In other words, while assigning TRUE to the returned variable forces the constraint to be satisfied, assigning FALSE to it forces the constraint to be not satisfied. It is worth noting that, conversely, assigning values to the variables of the constraints might lead, through propagation, to the assignment of values to the returned variables.

### 4.1.4 Theories

Compatibly with the DPLL(\(T\)) architecture proposed in [50] the `theory` abstract class needs to be notified, through the `propagate()` method, whenever some literal is assigned. The method replies with a boolean representing whether the theory propagation succeeded or some inconsistency is found. In case of failure, the `cnfl` vector is filled with the reason for the failure. Specifically, in case of conflict, this vector would be filled with a collection of assigned literals (possibly, a strict subset of all the assigned literals) that, together, would reproduce the conflict. An additional method `check()`, which might be less cheap than `propagate()`, checks for the consistency of the theory returning, similarly to the `propagate()` method, a boolean representing whether the theory is consistent. Again, in case of failure, the `cnfl` vector is filled with the reason for the failure. Finally, two methods `push()` and `pop()` notify the theory whenever a new backtracking point might be required or a backtracking step is performed.

```cpp
class theory {
    bool propagate(lit p, vector<lit> cnfl)
    bool check(vector<lit> cnfl)
    void push()
    void pop()
}
```

### 4.1.5 Propagation

The propagation algorithm keeps a set of literals (unit information) that is to be propagated into a queue called propagation queue. Whenever a literal is inserted into the queue the corresponding variable is immediately assigned. For each literal in the queue, the corresponding watcher list indicates the constraints which might be affected by the
assignment, therefore, their \textit{propagate()} method needs to be called in order to check
if more unit information might be inferred (and, consequently, enqueued). The process
continues until either the queue becomes empty or a conflicting clause is found.

As opposed to \textit{MiniSat} and, albeit with some adaptation, consistently with [50],
the original propagation procedure has been slightly adapted to consider theory propa-
gation.

```cpp
bool propagate(vector<lit> cnfl) {
    while (!prop_p.empty()) {
        lit p = prop_q.dequeue();
        vector<clause> tmp = move(watches[index(p)]);
        for (unsigned i = 0; i < tmp.size(); i++)
            if (!tmp[i].propagate(this, p)) {
                // the constraint is conflicting..
                for (unsigned j = i + 1; j < tmp.size(); j++)
                    watches[index(p)].add(tmp[j]);
                prop_q.clear();
                cnfl.add_all(tmp[i].lits)
                return false
            }
        // theory propagation..
        for (theory th : theories)
            if (!th.propagate(p, cnfl)) {
                prop_q.clear();
                return false
            }
    }
    // theory check..
    for (theory th : theories)
        if (!th.check(cnfl)) return false

    return true
}
```

The method for enqueuing information is rather straightforward. Specifically, \textit{enqueue()} puts a new fact into the propagation queue and immediately updates the variable’s value in the assignment vector. If a conflict arises, \textit{false} is returned. The parameter \textit{from} contains a reference to the constraint from which \textit{p} was propagated (defaults to \textit{null} if omitted). This is used for generating explanations when a conflict arise. Note that the same fact can be enqueued several times, as it may be propagated by several con-
straints, yet it is put in the propagation queue only once.

```cpp
bool enqueue(lit p, clause from = null) {
    switch(value(p)) {
    case True:
        return false
    case False:
        return true
    case Unassigned:
    // new fact, store it
        level[p.v] = trail_lim.size()
        reason[p.v] = from
        trail.add(p)
    }
Notice that the above method does not directly notify theories for new assignments. This task is, as we have seen, left to the `propagate()` method which, in case of need, can handle theory conflicts. More details about theory propagation will follow soon.

### 4.1.6 Learning

The final aspect worth to be investigated regards conflict-driven clause learning. This technique was first described in [83] and is considered one of the major advances of SAT technology in the last decades. The algorithm starts from a clause, called conflict clause, which is not satisfied by the current assignment. Let us assume this clause is, for example, \((\neg x_0 \lor x_1 \lor x_2)\). The set \(\{\neg x_0, \neg x_1, \neg x_2\}\) is called the reason of the conflict which is, basically, the assignments which make the clause unsatisfied. Suppose, for example, \(x_0\) is false becuase \(\neg x_0\) was propagated from some constraint, for example, \((\neg x_3 \lor \neg x_4 \lor \neg x_0)\). The reason for propagating \(\neg x_0\) is, therefore, \(\{x_3, x_4\}\). Indeed, \(x_3\) and \(x_4\) are the assignments which, through the clause \((\neg x_3 \lor \neg x_4 \lor \neg x_0)\), propagated \(\neg x_0\). From this little analysis it is possible to deduce that the set \(\{x_3, x_4, \neg x_1, \neg x_2\}\) would also lead to a conflict, hence, it is possible to prohibit this conflict by adding the learnt clause \((\neg x_3 \lor \neg x_4 \lor x_1 \lor x_2)\). In this example, only one literal (i.e., \(x_0\)) was analysed, however, the process of expanding literals can be iterated until all the literals of the conflict set are decision variables (i.e., whose level is lower than the current decision level).

It might be worth to further clarify this concept with a complete example. Let us consider the following set of clauses:

\[
\begin{align*}
  w_1 &= (x_1 \lor x_2) \\
  w_2 &= (x_1 \lor x_3 \lor x_7) \\
  w_3 &= (\neg x_2 \lor \neg x_3 \lor x_4) \\
  w_4 &= (\neg x_4 \lor x_5 \lor x_8) \\
  w_5 &= (\neg x_4 \lor x_6 \lor x_9) \\
  w_6 &= (\neg x_5 \lor \neg x_6)
\end{align*}
\]

Suppose that at the first decision level (i.e., root level), following some heuristic, the literal \(\neg x_7\) is assumed. The propagation procedure cannot deduce any further value for the other variables. The heuristic is then queried again and again assuming, for example, the literals \(\neg x_8\) and \(\neg x_9\), none of which propagates information. At this point the heuristic decides to assign the value FALSE to the variable \(x_1\). The propagation procedure, through the \(w_1\) clause, infer \(x_2 = \text{TRUE}\) and, combining the previous \(x_7 = \text{FALSE}\) assignment and the \(w_2\) clause deduces \(x_3 = \text{TRUE}\). The propagation procedure continues on this line generating the graph, called implication graph, of Figure 4.1 until \(x_5 = \text{FALSE}\) is deduced by clause \(w_6\). Since \(x_5\) cannot be both \(\text{TRUE}\), as a consequence
of the propagation of the clause $\omega_4$, and FALSE, because of $\omega_6$, the implication graph contains an inconsistency which is represented by the conflict clause $\omega_6$.

Notice that the implication graph does not need to be represented explicitly as a graph. An equivalent representation is given by the reason state variable containing, for each variable, the clause that propagated a value for the variable itself. Different learning schemas based on the above process as, for example, the “Last Unique Implication Point” (Last UIP) [5], have been proposed. Experimentally, the “First Unique Implication Point” (First UIP), detailed by the following code snippet, has been shown to be effective [113]. A further comparison between these techniques, in our specific case, however, would be desirable.

```c
void analyze(vector<lit> confl, vector<lit> out_learnt, unsigned out_btlevel) {
  out_btlevel = 0
  set<var> seen
  lit p = trail.last()
  vector<lit> p_reason = confl
  int counter = 0
  do {
    for (lit q : p_reason) {
      if (!seen(q.v)) {
        seen.add(q.v)
        if (level[q.v] == trail.lim.size()) {
          counter++
        } else if (level[q.v] > 0) {
          out_learnt.add(q)
          out_btlevel = max(out_btlevel, level[q.v])
        }
      }
    }
    do {
      p = trail.last()
      p_reason = reason[p.v].lits / reason[p.v].lits[0]
      pop_one()
    } while (!seen(p.v))
  }

Figure 4.1: Implication graph containing a conflict.
```
4.1. THE SAT CORE

```c
counter --
} while (counter > 0)
out_learnt[0] = ¬p
```

The idea behind the above algorithm is, simply, to walk breadth-first the implication graph starting from the conflict clause until a unique implication point, as close as possible to the conflict, is found. In doing so, the algorithm exploits the assignment order of the literals (i.e., the trail state variable), adding those visited literals which were assigned at a lower level excluding, for optimization reasons, those which were assigned at root level. The unique implication point, in the above example, is constituted by the node \( x_4 = \text{TRUE} \), being a unique node which, together with other previously assigned literals (i.e., \( x_8 \) and \( x_9 \)), would arouse the conflict. The learnt clause (i.e., \((¬x_4 \lor x_8 \lor x_9)\)) can be safely added to the set of clauses since it contains information which is deducible from the original set of clauses. Furthermore this clause, interfering with the propagation procedure, will avoid that the conflict will reappear again in different areas of the search space. Finally, the backtracking level, computed while analyzing the conflict, is the lowest decision level which makes the conflict clause unit, hence, we can backtrack until the learnt clause is applicable (i.e., any of the variables associated to its literals has UNASSIGNED value). Backtracking as far as possible is highly advantageous [83] and is usually referred, in the literature, as back-jumping or non-chronological backtracking.

Finally, the `record()` method stores a new learnt clause. Similarly to the `new_clause()` method, this method avoids the creation of new clauses in case the clause is unit. Since this method can, in principle, be called at any level, filtering on assigned literals is skipped.

```c
void record(vector<lit> lits) {
    if (lits.size() == 1) {
        enqueue(lits[0]) // cannot fail at this point..
    } else if (value(lits[0]) != Unassigned) {
        clause c = new clause(lits)
        enqueue(lits[0], c)
        constrs.add(c)
    }
}
```

4.1.7 Search

As mentioned earlier, the `sat_core` module does not implements a complete search algorithm. Search is, indeed, demanded to the `sat_core` user applications. Furthermore, given the specific nature of our problem, we do not want solutions having all the propositional variables assigned and, moreover, SAT based heuristics are not informative enough for solving our higher level problem. The `assume()` method, however, from the external interface, allows the users to assign TRUE to literals returning false if some immediate conflict is found (i.e., the literal is already FALSE).

```c
bool assume(lit p) {
    trail_lim.add(trail.size())
    for (theory th : theories)
```
In addition to the above `assume()` method, the `check()` method propagates clauses and, in case a conflict is found, it first calls the `analyze()` method for analyzing the conflict and generating a no-good, it then backtracks to the proper decision level and, finally, it adds the new learnt clause to the problem constraints. Finally, the `check()` method returns `true` if the propagation is successful or `false` if the problem is unsolvable. Notice that, the successfulness of propagation does not necessarily imply that a solution to the `sat_core` problem has been found. Specifically, further assignments and, consequently, checks, might be required to reach a solution state. On the contrary, in case the problem is recognized as unsolvable, it is possible to infer that there is no solution to the problem.

```cpp
bool check() {
    vector<lit> cnfl
    while(true) {
        if (!propagate(cnfl)) {
            if (root_level()) return false
            vector<lit> no_good
            unsigned bt_level
            analyze(cnfl, no_good, bt_level)
            while (trail_lim.size() > bt_level) {
                pop()
                record(no_good)
                cnfl.clear()
            } else
                return true
        }
    }
    return true
}
```

It is worth to notice that the above `check()` method might bring the `sat_core` at a lower level than the invoking decision level. As it will be shown soon, however, the `pop()` method will notify the bound theories.

The second `check()` method is aimed at assuming a given set of literals returning whether, after the assumptions and propagation, the problem has not detected inconsistencies. Before returning a value, however, this method is responsible for restoring the state as it was before the invocation.

```cpp
bool check(vector<lit> lits) {
    unsigned c_level = trail_lim.size()
    vector<lit> cnfl
    for (lit p : lits) {
        if (!assume(p)) || !propagate(cnfl)) {
            while (trail_lim.size() > c_level)
                pop()
            return false
        }
    }
    while (trail_lim.size() > c_level)
        pop()
    return true
}
```
4.2. THE LINEAR REAL ARITHMETIC (LRA) THEORY

It is worth to notice that the above check() procedure does not generate no-goods. It might be interesting to check whether analyzing the conflicts returned by the propagation procedure would return effective no-goods which would improve the performance of the resolution process. Nonetheless, allowing to backjump to a decision level that is lower than the one that is calling the procedure might make the software interface less intuitive.

Finally, the next two methods are utility methods for managing backtracking. Specifically, the aim of the pop() method is to revert to the state before the last push(). Additionally, pop() notifies the theories that sat_core is backtracking.

```c
void pop() {
    while(trail_limit.last() < trail.size())
        pop_one()
    trail_limit.pop()
    for (theory th : theories)
        th.pop()
}
```

The pop_one() method, instead, is used for undoing a single assignment. In other words, it simply unbinds the last variable on the trail.

```c
void pop_one() {
    lit p = trail.last()
    assigns[p.v] = Unassigned
    reason[p.v] = null
    level[p.v] = 0
    trail.pop()
}
```

4.2 The Linear Real Arithmetic (LRA) theory

How is it possible to reason on numerical constraints avoiding the use of (in our case) inefficient arc-consistency based propagation techniques? A possible alternative would be to exploit one of the different proposed techniques based on the Simplex method. Given our specific application, however, the reason for not using any of them is twofold: (a) most of the constraints, in our case, involve no more than two variables, against a number of variables that grows to tens of thousands (i.e., the matrix, built by the Simplex method, would be highly sparse) and (b) we need to add new variables and add new constraints, in case of assumptions, or remove existing constraints, when backtracking, easily, avoiding expensive copies of matrices. Usually, a tableau can be constructed and updated incrementally: rows are added and later removed when backtracking. These frequent addition and removal of rows (and the related bookkeeping), however, might have a significant cost. For example, backtracking may require pivoting operations. In other words, we need some kind of efficient incremental capabilities which reduces as much as possible this overhead.

The combined solution of the two previous problems is described in [37]. Specifically, the original arithmetic formula \( \Phi \) (i.e., a conjunction of arithmetic constraints) is transformed into an equisatisfiable formula \( \Phi_A \land \Phi' \) which is easily embeddable into the above sat_core module as a theory.
4.2.1 Preprocessing

The preprocessing step is aimed at rewriting the original arithmetic formula $\Phi$ into an equisatisfable formula $\Phi_A \land \Phi'$. Specifically, $\Phi_A$ is a conjunction of linear equalities and $\Phi'$ is a conjunction of elementary atoms of the form $x \vartriangleleft c$ where $x$ is a variable and $c$ is a rational constant ($\vartriangleleft$ is one of the operators $\leq$, $<$, $=$, $>$, $\geq$). The transformation is straightforward and can be shown through a simple example (used, also, in [37]).

Suppose we want to preprocess the following $\Phi$ formula:

$$x \geq 0 \land (x + y \leq 2 \lor x + 2y - z \geq 6) \land (x + y = 2 \lor x + 2y - z > 4)$$

Like in the Simplex method, it is possible to introduce two slack variables $s_1$ and $s_2$ and rewrite $\Phi$ as $\Phi_A \land \Phi'$:

$$(s_1 = x + y \land s_2 = x + 2y - z) \land x \geq 0 \land (s_1 \leq 2 \lor s_2 \geq 6) \land (s_1 = 2 \lor s_2 > 4)$$

which is, clearly, equisatisfable as $\Phi$. In general, starting from a formula $\Phi$, the transformation introduces a new slack variable $s_i$ for every linear term $t_i$ that is not already a variable occurring as the left side of an atom $t_i \vartriangleleft c$ of $\Phi$. Given this, $\Phi_A$ becomes the conjunction of all the equalities $s_i = t_i$ while $\Phi'$ is obtained by replacing all the terms $t_i$ by the corresponding $s_i$ in $\Phi$.

The formula $\Phi_A$ can be written in a matrix form as $Ax = 0$, where $A$ is a fixed $m \times n$ rational matrix, $m$ is the number of additional slack variables $s_1, \ldots, s_m$, $n$ is the number of initial variables $x_1, \ldots, x_n$ and $x$ is a vector in $\mathbb{R}^n$. The rows of $A$, by construction, are linearly independent so $A$ has rank $m$. The interesting aspect of this transformation consists in the fact that the formula $\Phi_A$ must always be satisfied, regardless of any transformation (i.e., pivoting) is performed on the $A$ matrix. On the contrary, each of the elementary atoms in $\Phi'$ can be associated (we say bound) to a propositional variable of the sat_core module which will decide if (and how, according to the assignment of the propositional variable) the elementary atom will be enforced or not. In other words, only the elementary atoms in $\Phi'$ are reified. Additionally, it is worth to notice that, overall, rewriting $\Phi$ as $\Phi_A \land \Phi'$ leads to problems with fewer variables compared to the standard Simplex method. Furthermore, some of the variables in $\Phi_A$ can be further simplified by applying Gaussian elimination.

In order to avoid the possibility of incurring in approximation issues, a new data structure is introduced for representing rationals. Specifically, both the numerator and the denominator are explicitly represented by means of two integer values. Additionally, arithmetic operators, not reported here for sake of space, are redefined so as to maintain the two numbers as normalized while handling special cases like $+\infty$ (i.e., $1/0$) and $-\infty$ (i.e., $-1/0$).

```c
class rational {
    long num // the numerator of the rational
    long den // the denominator of the rational
}
```

Finally, it is worth to notice that strict inequalities can be managed by introducing infinitesimals. As an example, a strict inequality like $x > 0$ can be rewritten ad $x > = \varepsilon$
4.2. THE LINEAR REAL ARITHMETIC (LRA) THEORY

where \( \varepsilon \) represents an infinitesimal value which is as small as needed. Rather than explicitly computing a value for the infinitesimal, however, it is possible to treat it symbolically. Numeric values could be represented, therefore, by means of pairs of rationals so as to represent values such as, for example, \( 1/3 + 1/5\varepsilon \).

The following operations and comparison, however, need to be defined on such values:

\[
(rat_0, inf_0) + (rat_1, inf_1) \equiv (rat_0 + rat_1, inf_0 + inf_1) \\
a \times (rat, inf) \equiv (a \times rat, a \times inf) \\
(rat_0, inf_0) \leq (rat_1, inf_1) \equiv (rat_0 < rat_1) \lor (rat_0 = rat_1 \land inf_0 \leq inf_1)
\]

where \( a \) is a rational number.

Both bounds and variable assignment now range on such values. Specifically, each of the involved numerical variable \( x_i \) is associated to three values: \( l_i \) (an \text{inf_rational}), representing the lower bound of the \( x_i \) variable, \( u_i \) (another \text{inf_rational}), representing the upper bound of the \( x_i \) variable and \( \beta_i \) (yet another \text{inf_rational}), initially set to 0, representing the current assigned value of the \( x_i \) variable.

4.2.2 The Application Programming Interface (API)

We start by presenting the linear real arithmetic theory’s external interface, with which a user application can specify, along with the \text{sat_core} module, linear arithmetic problems.

```csharp
class inf_rational {
    rational rat // the rational part..
    rational inf // the infinitesimal part..
}
```

Similarly to the \text{sat_core} module, new variables are introduced by calling the \text{new_var()} procedure. \text{var} is, analogously, a type synonym for an \text{unsigned int}. Unlike the \text{sat_core} module, however, rather than a propositional variable, \text{new_var()} returns an arithmetic variable. Such variables can be combined, together with coefficients and constants, to generate the \text{lin} linear expressions which, in turn, can be combined to form constraints. Specifically, the \text{leq()} and the \text{geq()} methods are used generate the reified \( \leq \) and \( \geq \) constraints which are represented by means of the returned propositional variables which, in turn, are bound to the \text{sat_core} module. Enforcing (or negating) such constraints can be done by assigning these propositional variables the \text{TRUE} (or \text{FALSE}) value, through the \text{sat_core} module, either by means of propositional constraints (e.g., clauses) or by means of direct assumptions (i.e., the \text{assume()}
method of `sat_core`). Finally, given an arithmetic variable `x`, the `bounds()` and `value()` methods return the bounds (an `interval`, as will soon be explained, is a basic data type for representing, together, a lower and an upper bound) and the current value of the `x` variable.

Before proceeding further, it is worth to introduce the basic data structures needed for implementing our linear arithmetic theory and to bind it to the `sat_core` module. Specifically, the `lin` data structure is used for representing linear expressions.

```cpp
class lin {
    map<var, rational> vars
    rational known
}
```

Since this Simplex method implementation relies on the Bland’s pivot-selection rule to ensure termination, which in turn relies on a total order on variables, the above map is implemented through a Red-Black tree which, by construction, maintains the map keys ordered according to their default ordering. Furthermore, arithmetic operators (i.e., `+`, `−`, `×` and `÷`) are overridden, while preserving linearity (i.e., avoiding the introduction of non-linear expressions), in order to facilitate the combination of linear expressions.

The `interval` class, on the other hand, simply represents the bounds of a variable. Such bounds are initialized, respectively, to `−∞` and to `∞`.

```cpp
class interval {
    inf_rational lb
    inf_rational ub
}
```

Even in this case, arithmetic operators (i.e., `+`, `−`, `×` and `÷`), together with some boolean relations (e.g., `≤`, `<`, `=`, `>`, `≥`, etc.) are overridden allowing a subset of interval algebra over such intervals.

### 4.2.3 The solver state

Similarly to the `sat_core` case, also in the linear arithmetic solver state a number of things need to be stored:

```cpp
sat_core sat // the sat core..
// the current bounds (i.e., lower and upper bounds)..
vector<bound> bounds
vector<inf_rational> vals // the current values..
// the expressions (string to numeric variable) for which already
// exist slack variables..
map<string, var> exprs
// the assertions (string to propositional variable) used for reducing
// the number of propositional variables..
map<string, var> s_asrts
// the assertions (propositional variable to assertion) used for
// enforcing (negating) assertions..
map<var, assertion> v_asrts
// for each variable `v`, a list of assertions watching `v`..
vector<vector<assertion>> a_watches
// for each variable `v`, a list of tableau rows watching `v`..
vector<vector<row>> t_watches
```
4.2. THE LINEAR REAL ARITHMETIC (LRA) THEORY

Specifically, the \textit{sat} field is a reference to the \textit{sat\_core} needed for creating and binding new propositional variables. The \textit{bounds} field, relying on the \textit{bound} data structure described later, stores the lower and the upper bound, as well as the reason for having the bound, for each of the arithmetic variables and, thus, allow reification of constraints. As will be shown soon, it is possible to retrieve these bounds by means of two utility functions which translate a \textit{var} to its respective lower and upper bound index. Additionally, the \textit{vals} field stores, for each arithmetic variable, its current value. The elements stored within the \textit{exprs} map represent all the already seen expressions. Functional to the transformation of the original linear arithmetic formula $\Phi$ to the equisatisfiable formula $\Phi_A \land \Phi'$, this map has string representations of the expressions as keys and arithmetic variables as values. The \textit{s\_asrts} and the \textit{v\_asrts} maps, on the other hand, are used for binding elementary atoms, hereafter called \textit{assertions}, with propositional variables of the \textit{sat\_core} module. Specifically, the \textit{s\_asrts} uses string representations of the assertions as keys and propositional variables as values. This map is used for avoiding the creation of different propositional variables associated to the same assertion. The \textit{v\_asrts} map, conversely, is used for asserting atoms whenever the \textit{propagate()} method of the theory is called by the \textit{sat\_core} module. When the corresponding literal becomes \textit{True}, this map is used for retrieving the proper assertion so as to allow its enforcement. In a nutshell, the \textit{s\_asrts} and the \textit{v\_asrts} maps represent, together, the $\Phi'$ formulas. Similarly to the \textit{sat\_core}’ \textit{watches}, the \textit{a\_watches} and the \textit{t\_watches} vectors are used for maintaining, for each variable \textit{v}, respectively, the list of assertions and the list of of tableau raws watching \textit{v}. As will be shown soon, these watch-lists are used for implementing theory propagation. Finally, the \textit{layers} field, for each decision level, stores a map having the index of the bound as key and the \textit{bound} as value.

In addition to the above state variables, the solver state includes a tableau derived from the $A$ matrix which can be written in the form

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \quad x_i \in B$$

where $B$ and $\mathcal{N}$ denote the set of basic and non-basic variables, respectively. Additionally, $B$ corresponds, initially, to the set of the newly introduced slack variables $s_1, \ldots, s_m$. The tableau is practically represented through the following map.

\begin{verbatim}
// the (sparse) matrix
map<var, row> tableau
\end{verbatim}

In a nutshell, the \textit{tableau} map is used for representing the $\Phi_A$ equalities. As already said, this Simplex method implementation relies on the Bland’s pivot-selection rule to ensure termination, hence we need a total order of the keys of the above map. Such a total order can be achieved efficiently by means of Red-Black trees. Notice that by using maps, both in the tableau and in the linear expressions, we can significantly reduce the size of the matrix while preserving the efficient introduction of new rows as well as the efficient mathematical operations (i.e., pivoting).
A distinguishing factor between basic and non-basic variables consists in the fact that the values assigned to the non-basic variables must always be consistent with their bounds. This is not the case for basic variables which, in some cases, might have values outside of their bounds. The check() procedure, described later, will be responsible for adjusting these temporary flaws. By contrast, the assigned values, both for basic and non-basic variables, are always maintained consistent through the $\Phi_A$ and $\Phi'$ formulas.

For the sake of completeness, the bounds are stored by means of the following data structure which allows to store, together, the value of the bound and, represented through a literal, its reason.

```java
class bound {
    inf_rational val
    lit reason
}
```

Finally, the following helper methods are defined for readily retrieving the bounds and the current values of both variables and linear expressions.

```java
inf_rational lb(var x) { return bounds[lb_index(x)] }
inf_rational ub(var x) { return bounds[ub_index(x)] }
interval bounds(var x) { return interval(lb(x), ub(x)) }
interval bounds(lin l) {
    interval bs = l.known
    for (var x : l.vars.keys) { bs += l.vars[x]*bounds(x) }
    return bs
}
inf_rational value(var x) { return vals[x] }
inf_rational value(lin l) {
    inf_rational val = l.known
    for (var x : l.vars.keys) { val += l.vars[x]*value(x) }
    return val
}
unsigned lb_index(var v) { return v << 1 }
unsigned ub_index(var v) { return (v << 1) ^ 1 }
```

### 4.2.4 Variables and constraints

The following code snippet is aimed at creating arithmetic variables. The code initializes the bounds and the current value of the new variables. In addition, it creates a watch list for theory propagation and stores a new expression, consisting of the sole variable, within the exprs map.

```java
var new_var() {
    var x = bounds.size()
    bounds.add(bound(-inf, null)) // the lower bound..
    bounds.add(bound(inf, null)) // the upper bound..
    vals.add(0)
    exprs["x" + x] = x
    a_watches.add(set<assertion>())
    t_watches.add(set<row>())
    return x
}
```
Since there are two kinds of formulas (i.e., $\Phi_A$ and $\Phi'$) there are also two kinds of linear constraints. Specifically, we use *rows* to represent the tableau rows. These constraints are used to represent the $\Phi_A$ formula and are always satisfied. Additionally, we use *assertions* to represent the relations between variables and values within the $\Phi'$. Such assertions are, to all effects, reified constraints and are, hence, associated to a propositional variable. Notice that the satisfaction of an assertion can be either imposed by the *sat_core*, by invoking the theory’s propagate method during propositional propagation, or detected by the theory itself if, as a consequence of imposing an assertion, the bounds of a variable taking part in another assertion become such as to establish, incontrovertibly, that the other assertion is satisfied or not.

As already mentioned, the *row* class is a specific linear constraint aimed at representing tableau rows. In particular, the $x$ variable represents the basic variable while the $l$ field is used to represent the associated linear expression. Instances of these classes are stored within the $t\_watches$ map so as to allow theory propagation by means of the `propagate_lb` (and `propagate_ub`) methods. Specifically, these methods notify the constraint that the lower bound (respectively, upper bound) of a variable has changed. This change might lead to the change of a bound of the basic variable $x$ which, if taking part in an assertion, might propagate information to the *sat_core* regarding the (un)satisfaction of the assertion. In case an inconsistency is detected, the $cnfl$ vector is filled with an explanation of the inconsistency. More details about theory propagation will follow later.

class row {
    row(la_theory th, var x, lin l) {
        for (var x : l.vars) { th.t\_watches[x].add(this) }
    }
    bool propagate_lb(var x, vector<lit> cnfl)
    bool propagate_ub(var x, vector<lit> cnfl)
    var x // the basic variable..
    lin l // the linear expression..
}

The *assertions*, on the other hand, are aimed, as already briefly mentioned, at representing relations between variables and constants (i.e., at imposing bounds on the variables’ domains). Instances of these classes are stored within the $s\_asrts$ (actually, only a string representation of the assertion is stored within the $s\_asrts$ map) and within the $v\_asrts$ map so as to represent the $\Phi'$ set of elementary atoms.

class assertion {
    assertion(la_theory th, op o, var b, var x, inf\_rational v) {
        th.a\_watches[x].add(this)
    }
    bool propagate_lb(var x, vector<lit> cnfl)
    bool propagate_ub(var x, vector<lit> cnfl)
    op o // the operator kind..
    var b // the propositional variable..
    var x // the arithmetic variable..
    inf\_rational v // the constant..
CHAPTER 4. THE SMT-BASED CONSTRAINT NETWORK

This data structure stores the propositional variable (i.e., $b$) bound with the theory, the arithmetic variable (i.e., $x$) and the constant value with which the arithmetic variable is constrained (i.e., $v$). The operator $op \in \{\leq, \geq\}$ represents the relation between the arithmetic variable and the constant value. This class is used both for asserting atoms and for performing unate propagation. Roughly speaking, in case of a $\leq$ assertion, as soon as $\text{TRUE}$ (or $\text{FALSE}$) is assigned to the propositional variable $b$, the $x \leq v$ ($x > v$) assertion is enforced. Conversely, as soon as the constraint becomes asserted (i.e., $ub \leq v$) or negated (i.e., $lb > v$), as a consequence of theory propagation, a new literal (i.e., $b$ or $\neg b$) is enqueued into the $\text{sat_core}$ module.

Regarding the implementation of the methods for creating the constraints, they are, conceptually, all alike and, therefore, only one will be shown. The implementation of the $\leq$ constraint is quite simple. At first, a new linear expression $expr$ is created as the difference of the given expressions. Afterwards, all of its basic variables are replaced with their corresponding linear expressions from the tableau. Subsequently, by means of the $\text{mk_slack()}$ method, the resulting linear expression, deprived of the known term, is then equalled with a slack variable enriching the set of equalities of $\Phi_A$. Finally, the slack variable is constrained, accordingly with the specific constraint method, with the negated known term of the linear expression. Notice that comparing the negated of the resulting linear expression’s known term with the expression itself, deprived of the known term, can, in some cases, avoid the creation of new variables.

```plaintext
var leq(lin l, lin r) {
    lin expr = left − right
    for (var x : expr.vars.keys) {
        if (tableau.find(x)) {
            rational c = expr.vars[x]
            expr.vars.erase(x)
            expr += tableau[x] * c
        }
    }

    inf_rational val = −expr.known
    expr.known = 0
    interval i = bounds(expr)
    if (i <= val)
        return ⊤/\ the constraint is already satisfied\.
    else if (i > val)
        return ⊥/\ the constraint is unsatisfable\.

    var s = mk_slack(expr)
    string asrt = "x" + s + " <= " + val
    if (s.asrsts.find(asrt))
        return s.asrsts[asrt]
    else {
        var ctr = sat.new_var()
        s.asrsts[asrt] = ctr
        assertion a = new assertion(this, le, ctr, s, val)
        v.asrsts[ctr].add(a)
        sat.bind(ctr, this)
        return ctr
    }
}
```
It is worth noting that in case the inequality is strict, the val value involves an infinitesimal and is, hence, created considering both the negation of the expression’s known term and a negative infinitesimal as described by the following code snippet.

```plaintext
inf_rational val = inf_rational(-expr.known, -1)
```

Starting from a linear expression, the utility method `mk_slack()` is responsible for creating a new slack variable or, whenever possible, to reuse an already existing variable. Thanks to the `exprs` map, indeed, it is often possible to reuse existing expressions without the need to create new slack variables. Notice that, in case the given linear expression is such that a new slack variable needs to be created, the current value for the new variable is initialized accordingly to the values of the linear expression and a new row is added to the tableau.

```plaintext
var mk_slack (lin l) {
    string s_expr = to_string(l)
    if (exprs.find(s_expr))
        return exprs[s_expr]
    else {
        var slack = new_var()
        exprs[s_expr] = slack
        vals[slack] = value(l)
        tableau[slack] = new row(slack, l)
        return slack
    }
}
```

It is worth noting that implementing an equality constraint between two linear expressions is straightforward and can be realized as a conjunction of $\leq$ relation and of $\geq$ one (e.g., $x_0 == x_1 \equiv x_0 \leq x_1 \land x_0 \geq x_1$). Finally, since the equality constraint returns, as any reified constraints, a propositional variable, realizing a disequality constraint (e.g., $x_0 \neq x_1$) can be realized through the negation of the propositional variable returned by the equality constraint.

### 4.2.5 Updating variables and pivoting the tableau

The following code snippets represents two helper methods which are used to handily modify $\beta$. Specifically, `update()` sets the value of the non-basic variable $x_i$ to $v$ and updates the values of the basic variables, which have some connection to $x_i$, so that all the equations remain satisfied.

```plaintext
void update(var x_i, inf_rational v) {
    for (row r : t_watches[x_i])
        vals[r.v] += r.l.vars[x_i]*v - vals[x_i]
    vals[x_i] = v
}
```

The purpose of `pivot_and_update()` is to apply pivoting to the basic variable $x_i$ and the non-basic variable $x_j$. Additionally, it sets the value of $x_i$ to $v$ and updates the values of the basic variables, which have some connection to $x_i$ within the tableau, so that all the equations remain satisfied.
Finally, the \texttt{pivot()} procedure makes a simple pivoting operation between a basic variable \(x_i\) (since this basic variable leaves the base, it is also called \textit{exiting} variable) and a non-basic variable \(x_j\) (since this non-basic variable enters in the base, it is also called \textit{entering} variable).

\begin{verbatim}
void pivot_and_update(var x_i, var x_j, inf_rational v) {
    inf_rational \( \theta \) = \( (v - \text{vals}[x_i]) / \text{tableau}[x_i].l.vars[x_j] \)
    vals[x_i] = v
    vals[x_j] += \( \theta \)
    for (row r : t.watches[x_j])
        vals[r.v] += r.l.vars[x_j]*\( \theta \)
    pivot(x_i, x_j)
}
\end{verbatim}

It is worth to notice that the above \texttt{pivot()} procedure is also responsible for maintaining updated the watch-lists so as to allow theory propagation.

4.2.6 The \texttt{check()} procedure

As already mentioned, the main procedure is based on the Simplex method and relies on Bland’s pivot-selection rule to ensure termination. Specifically, it relies on a total order on variables. Such an ordering can be efficiently maintained, as already mentioned,
by making use of Red-Black tree based maps for representing both the tableau (using the basic variables as key and the linear expression as value) and the linear expressions (using the involved variables as key and their coefficients as values). The main procedure searches for a basic variable whose current value is outside of its bounds. If no such variable is found, the procedure terminates successfully. If such a variable is found, new assignments are produced and a new basis is found so that all the bounds are satisfied. Finally, if it is not possible to find such values, an unsatisfable constraint is found and used to generate an explanation for the conflict. The code for the check procedure is the following.

```cpp
bool check(vector<lit> cnfl) {
    while (true) {
        var xi = find_if(x ∈ B, vals[x] < lb(x) | vals[x] > ub(x))
        if (xi == null) { return true }
        if (vals[xi] < lb(x)) {
            var xj = find_if(x ∈ N,
                (tableau[xi][xj] > 0 & vals[xj] < ub(xj)) | (tableau[xi][xj] < 0 & vals[xj] > lb(xj)))
            if (xj == null) {
                for (var x ∈ N+)
                    cnfl.add(!bounds[ub_index(x)].reason)
                for (var x ∈ N-)
                    cnfl.add(!bounds[lb_index(x)].reason)
                return false
            }
            pivot_and_update(xi, xj, lb(x))
        }
        if (vals[xi] > ub(x)) {
            var x_j = find_if(x ∈ N,
                (tableau[xi][x_j] < 0 & vals[x_j] < ub(x_j)) | (tableau[xi][x_j] > 0 & vals[x_j] > lb(x_j)))
            if (x_j == null) {
                for (var x ∈ N+)
                    cnfl.add(!bounds[ub_index(x)].reason)
                for (var x ∈ N-)
                    cnfl.add(!bounds[lb_index(x)].reason)
                cnfl.add(!bounds[ub_index(xi)].reason)
                return false
            }
            pivot_and_update(xi, x_j, ub(x))
        }
    }
}
```

where \(\mathcal{N}^+\) represents those non-basic variables, in a row whose basic variable’s value violates its bounds, whose coefficients are greater than 0 while \(\mathcal{N}^-\) represents, conversely, those non-basic variables whose coefficients are lower than 0.

The idea behind generating explanations consists in finding a basic variable whose value is not consistent with its bounds. Furthermore, the value of the non-basic variables of the corresponding row of the tableau are already at their bounds, hence, such values cannot be adjusted without violating other bounds. The literals which established the bounds of these non-basic variables, together with the literal that established
the violated bound of the basic variable, represent the unsat core, i.e., the set of literals which explain the inconsistency. Demonstrating that such an explanation is minimal is easy, yet, out of the scope of this document.

4.2.7 Theory propagation

An important aspect to consider when building a theory is theory propagation. In case of the linear real arithmetic theory, specifically, propagation results useful only in case if it can be done cheaply. As highlighted in [37], indeed, full propagation is just too expensive and results to be worthless. Nonetheless, no propagation is also a poor choice. Given a set of elementary atoms $A$ from the formula $\Phi'$, unate propagation is very cheap to implement. For example, if a lower bound $x_i \geq c$ has been asserted, then, any unassigned atom of $A$ of the form $x_i \geq c'$ with $c' < c$ is immediately implied. Similarly, the negation of any atom $x_i \leq c''$, with $c'' < c$, is also implied. Furthermore, in case these procedures are refining the bounds of non-basic variables, the bounds of the basic variables can be deduced by the bounds of the non-basic variables through a bound refinement procedure. These computed bounds may imply unassigned elementary atoms in $A$. Notice that these computed bounds are often weaker than the current bounds. Nonetheless, these kinds of propagations are very cheap to implement and in practice, often, result to be useful.

Taking inspiration by the watched literals of the sat-core, the a_watches and the t_watches fields keep track of those assertions and tableau rows which watch on variables. Specifically, whenever a watched variable updates its bounds, the constraints in its associated watched list are checked to see if some information might be propagated.

Propagating atoms, however, represents just one of the aspects of theory propagation which relies on the enqueue() method of the sat-core module. Specifically, unless the theory is at root level, the reason (i.e., a propositional constraint) for propagating any atom must be generated and sent to the enqueue() method. Since the analyze() method of the sat-core module strongly relies on such reasons, generating them is mandatory in order to produce explanations whenever an inconsistency is detected.

The propagate() function is called by the sat-core module both for notifying the theory of new assignments and for inducing theory propagation. The function implementation is the following.

```c
bool propagate(lit p, vector<lit> cnfl) {
    assertion a = v_asrts[p.v]
    switch(a.o) {
        case leq:
            if (p.sign)
                return assert_upper(a.x, a.v, p, cnfl)
            else
                return assert_lower(a.x, a.v, p, cnfl)
        case geq:
            if (p.sign)
                return assert_lower(a.x, a.v, p, cnfl)
            else
                return assert_upper(a.x, a.v, p, cnfl)
    }
}```
Specifically, the function relies on the following two basic procedures for updating the bounds of a variable. These procedures have no effect if the imposed bounds are weaker than the current ones. Conversely, if the new bounds generate an inconsistency (e.g., they impose, for a given variable, an upper bound to be lower than a lower bound or, on the contrary, a lower bound greater than an upper bound) a minimal explanation must be generated. This explanation consists, straightforwardly, of the negation of the literal whose assignment gave rise to the propagation together with the negation of the literal that, previously, imposed the conflicting bound. Since a bound can, in general, be updated several times within the same decision level, it is stored only the first time it is updated. The `assert_upper()` method works just the opposite way and is not reported for sake of space.

```plaintext
bool assert_lower(var x, inf_rational c, lit p, vector<lit> cnfl) {
    if (c <= lb(x)) return true
    if (c > ub(x)) {
        // either the literal 'p' is false ..
        cnfl.add(!p)
        // .. or what asserted the upper bound is false..
        cnfl.add(!bounds[ub_index(x)].reason)
        return false
    }
    // stores the current bound for subsequent backtracking..
    if (!layers.empty() && !layers.top().contains(x)) { layers.top().lb[x] = c }
    bounds[lb_index(x)] = bound(c, p)
    if (x ∈ N & vals[x] < c) update(x, c)
    // theory propagation..
    for (assertion asrt : a_watches[x])
        if (!asrt.propagate_lb(x, cnfl)) return false
    for (row r : t_watches[x])
        if (!r.propagate_lb(x, cnfl)) return false
    return true
}
```

As can be seen from the previous code, most of the complexity of theory propagation is demanded to the `propagate()` procedures of the assertions and of the tableau rows. In case of assertions, the first aspect to notice is that, either the assertion represents a ≤ relation or a ≥ one, the propagation of a lower bound update can take place only if the lower bound of the variable is strictly greater than the value of the assertion. This is because the bounds of the variables, already verified within the `assert_lower()` and `assert_upper()` procedures, are not in dispute per se. The propagated information is, conversely, related to the value of the variable b of the assertion.

In order to allow a better understanding of the theory propagation, the update of the lower bound of a variable x in an x ≥ c assertion, such that the lower bound becomes greater than c, is represented in Figure 4.2. Initially, the lower bound of the x is lower than the c constant of the ≤ assertion. The upper bound of the same variable, however, is greater than c making it impossible to determine if the assertion is satisfied or not. As
CHAPTER 4. THE SMT-BASED CONSTRAINT NETWORK

Figure 4.2: Propagating a lower bound update of $x$ such that $lb(x) > c$ in an $x \leq c$ assertion. The propagation results in assigning to the variable $b$ the value $\text{FALSE}$, in case of a $\leq$ assertion, and the value $\text{TRUE}$, in case of a $\geq$ assertion.

A consequence the update of the lower bound of $x$, the procedure is called and, because the lower bound has now become greater than $c$, we can infer that the assertion is unsatisfiable and, hence, $b = \perp$ is propagated. Several cases might occur: in case $b$ can be made $\text{FALSE}$ (i.e., its current value is $\text{Unassigned}$), propagation succeeds and a reason for the new value of $b$ must be provided; similarly, in case $b$ cannot be made $\text{FALSE}$ (i.e., its current value is $\text{TRUE}$), propagation fails and a reason for the conflict must be provided; finally, in case $b$ is already $\text{FALSE}$, no information is propagated. Notice that the literal that enforced the current lower bound of the $x$ variable, referred as $p$, represents both the reason for the propagation and part of the explanation for the conflict, beside being the last literal, until now, propagated by the $\text{sat\_core}$.

The following procedure describes the propagation of a lower bound change for a variable within an assertion.

```c
bool propagate_lb(var x, vector<lit> cnfl) {
    if (th.lb(x) > v) {
        switch(o) {
            case le:
                switch(sat.value(b)) {
                    case True:
                        // inconsistency...
                        cnfl.add(lit(b, false))
                        cnfl.add(!th.bounds[lb_index(x)].reason)
                        return false
                    case Undefined:
                        // propagation is safe...
                        th.record(lit(b, false), !th.bounds[lb_index(x)].reason)
                        break
                    case ge:
                        ...
                }
            break
        }
    }
    return true
}
```

The $\geq$ case is symmetrical since, conversely to the $\leq$ case, the assertion becomes satisfied and, as a consequence, rather than $b = \text{FALSE}$, $b = \text{TRUE}$ is propagated. This is
4.2. THE LINEAR REAL ARITHMETIC (LRA) THEORY

reflected in the polarity of the $b$ variable which, both in the conflict and in the reason for the propagation, appears directed. Given its simplicity, this case will not be described.

The propagation of the bounds, in the case of the rows of the tableau, takes place in a similar way. In case of tableau rows, however, more variables are involved and, computing the reason for the propagation or the cause for a conflict might result to be slightly more cumbersome. The following procedure describes the propagation of a lower bound change for a variable involved in a tableau row.

```cpp
bool propagate_lb(var v, vector<lit> cnfl) {
    cnfl.add(lit(0, false)) // make room for the first literal..
    if (l.vars[v] > 0) {
        inf_rational lb = 0 // the lower bound of the basic variable..
        for (entry term : l.vars) {
            if (term.second > 0) {
                if (th.lb(v) == -\infty) {
                    // nothing to propagate..
                    cnfl.clear()
                    return true
                } else {
                    lb += term.second * th.lb(term.first);
                    cnfl.add(!th.bounds[lb_index(term.first)].reason)
                }
            } else {
                ...
            }
        }
        if (lb > th.lb(x)) {
            // lb is a better bound..
            // we check if we can propagate some assertion..
            for (assertion a : th._a_watches[x]) {
                if (lb > a.v) {
                    switch(a.o) {
                        case le:
                            cnfl[0] = lit(c.b, false);
                            switch(th.sat.value(b)) {
                                case True:
                                    // inconsistency..
                                    return false
                                case Undefined:
                                    // propagation is safe..
                                    th.record(cnfl)
                                break
                                case ge:
                                    ...
                                }
                            }
            }
        }
    } else {
        ...
    }
    cnfl.clear()
    return true
}
```
4.2.8 Backtracking

The last aspect to consider for the linear real arithmetic theory regards the data structures and the procedures for storing the context and allowing backtracking. Since the number of backtracks is often very large, efficient backtracking is, indeed, important. As explained in [37], the strengths of the above proposed approach relies in the need to restore the sole bounds of updated variables as well as the reason for having updated them. Backtracking does not require saving the current values of variables before updating since the updated values must be compatible with the bounds of lower levels and, most interestingly, does not require to revert the performed pivotings.

It is worth to recall that, whenever a literal is assumed, through the assume() method of the sat_core, the push() method of every theory is called, allowing theories to store context for subsequent backtracking. The real linear arithmetic theory, simply, adds a layer whenever its push() method is called. By doing so, the assert_lower() and assert_upper() procedures are enabled to store, before updating, the bounds.

```c
void push() { layers.add(map<unsigned, bound>()) }
```

Finally, the pop() method of the sat_core calls the pop() method of every theory for restoring the context of the theory before push() was called. In our case, we simply restore the updated bounds.

```c
void pop() {
    for (entry b : layers.top()) { bounds[b.first] = b.second }
    layers.pop()
}
```

4.3 The Object Variables theory

A second theory, which will come back useful, concerns the management of object variables and the constraints among them. Strongly relying on the sat_core, the Object Variables (OV) theory is responsible for creating object variables, having a set of values as initial domain, and enforcing equality (i.e., =) constraints between them. In particular, the overall idea underlying the object variables theory consists in associating a propositional variable to each of the allowed values of each object variable and enforcing an exactly-one constraint among them. These variables will indicate whether the corresponding allowed value belongs to the domain of the variable.

The object variable theory’s external interface, with which a user application can specify, along with the sat_core module, problems, is described by the following code:

```c
class ov_theory : theory {
    var new_var(set<value> vs)
    var eq(var l, var r)
    var allows(var x, value v)
    set<value> value(var x)
}
```

Specifically, new variables are introduced by calling the new_var() procedure taking, as a parameter, the set of allowed values representing the initial domain. value, in
4.3. THE OBJECT VARIABLES THEORY

particular, is nothing more than a marker interface. \texttt{var} is, again, a type synonym for an \texttt{unsigned int}. In the case of this theory, however, the \texttt{new\_var()} procedure returns an object variable. Such variables can be used for generating equality constraints, by means of the \texttt{eq()} procedure, which are represented by propositional variables which, in turn, are bound to the \texttt{sat\_core} module. Enforcing (or negating) such constraints can be done by assigning these propositional variables the TRUE (or FALSE) value, through the \texttt{sat\_core} module, either by means of propositional constraints (e.g., clauses) or by means of direct assumptions (i.e., the \texttt{assume()} method of \texttt{sat\_core}). Given an object variable \(x\) and a value \(v\), the \texttt{allows()} procedure returns a propositional variable indicating whether the \(v\) value can be assumed by \(x\) the variable. Finally, given an \(x\) object variable, the \texttt{value()} procedure returns the set of allowed values associated to the \(x\) variable.

The following things need to be stored for representing the state:

```c++
//sat_core
sat // the sat core...
// the current assignments (i.e., the allowed values)...
vector<map<value, var>> assigns
// the expressions (string to object variable) for which already
// exist slack variables...
map<string, var> exprs
```

In particular, the \texttt{sat} field is a reference to the \texttt{sat\_core} needed for creating and binding new propositional variables. The \texttt{assigns} field stores, for each object variable, a mapping between each of its allowed values and the corresponding propositional variable describing the presence of the value within the domain of the object variable. Finally, the elements stored within the \texttt{exprs} map represent all the already seen expressions.

4.3.1 Variables and constraints

The following code snippet is aimed at creating new object variables. The code initializes the initial domain of the new variables assigning, for each of its allowed values, a propositional variable and enforcing, among them, an exactly-one constraint.

```c++
var new_var(set<value> vs) {
    var x = assigns.size()
    assigns.add(map<value, var>())
    if (vs.size() == 1)
        assigns[x][vs[0]] = T
    else {
        vector<lit> lits
        for (value v : vs) {
            var bv = sat.new_var()
            assigns[x][v] = bv
            lits.add(bv)
            bind(bv)
        }
        exact_one(lits)
    }
    return x
}
```
where the `exct_one()` procedure represents a utility method for enforcing an exactly-one constraint among the given literals. Notice that, conversely to the `sat_core`'s `new_exct_one()` procedure, this procedure does not create a reified constraint and, hence, does not return a propositional variable. Similarly to the `sat_core` case, however, the procedure generates a naive encoding, combining the at-most-one and the at-least-one constraint, requiring the introduction of $O(n^2)$ new clauses. For sake of completeness, the following code snippet describes the `exct_one()` procedure:

```c
void exct_one(vector<lit> lit) {
    vector<lit> tmp
    for (unsigned i = 0; i < lits.size(); i++) {
        for (unsigned j = i + 1; j < lits.size(); j++) {
            // the at-most-one constraints...
            sat.new_clause( {¬lits[i], ¬lits[j]} )
        }
        tmp.add(lits[i])
    }
    // the at-least-one constraint...
    sat.new_clause(tmp)
}
```

Suppose, for example, we want to create two object variables $o_0 \in \{a, b\}$ and $o_1 \in \{b, c\}$. By calling the `new_var()` procedure a first time, two new propositional variables $b_0$ and $b_1$, associated to the $a$ and $b$ allowed values, are created and the clauses $(¬b_0, ¬b_1)$, representing the at-most-one constraint, and $(b_0, b_1)$ representing the at-least-one constraint, are enforced so as to guarantee the exactly-one constraint between $b_0$ and $b_1$. Analogously, by calling the `new_var()` procedure a second time, two new propositional variables $b_2$ and $b_3$ are created and the $(¬b_2, ¬b_3)$ and $(b_2, b_3)$ constraints are enforced.

As regards the procedure for creating the equality constraint, its role can be reduced at enforcing the equality, subject to the controlling propositional variable, of the propositional variables corresponding to the same values. The following code snippet describes the equality constraint:

```c
var eq(var l, var r) {
    if (l == r) return $\top$
    if (l > r) return eq(r, l)

    string asrt = "e" + l + " == e" + r
    if (exprs.find(asrt))
        return exprs[asrt]
    else {
        set<value> intrsct
        for (value v : assigns[l].keys)
            if (assigns[r].find(v))
                intrsct.add(v)
        if (intrsct.empty()) return $\bot$

        var ctr = sat.new_var()
        exprs[asrt] = ctr

        for (value v : assigns[l].keys)
            if (!intrsct.find(v))
                sat.new_clause(lit(ctr, false), lit(assigns[v], false)))
```
Specifically, after a preprocessing phase, aimed at avoiding creating several times the same constraint, the intersection \( \text{intrsct} \) of the allowed values of the two variables is computed. In case the intersection is empty, the two object variables cannot be equal and, therefore, the \( \bot \) \( \text{var} \) variable is returned, otherwise, a new propositional variable \( \text{ctr} \) is created. Constraints are added so that, as a consequence of making the \( \text{ctr} \) variable \( \text{TRUE} \), the propositional variables, associated to those values which are not within the intersection, are constrained to be \( \text{FALSE} \) while the propositional variables, associated to those values which are within the intersection, are constrained to be pairwise equal.

As an example, creating the \( o_0 = o_1 \) constraint results in the creation of a new propositional variable \( b_4 \) and in the enforcement of the constraints \( \neg b_4, \neg b_0, \neg b_3, \neg b_4, \neg b_1, b_2, \neg b_3, b_1, \neg b_2 \) and \( b_4, \neg b_1, \neg b_2 \) and returns \( b_4 \). Assigning the value \( \text{TRUE} \) to \( b_4 \) would result in the assignment of the value \( \text{FALSE} \) to both \( b_0 \) (as a consequence of the propagation of the first constraint) and \( b_3 \) (as a consequence of the propagation of the second constraint) and in the assignment of the value \( \text{TRUE} \) to both \( b_1 \) and \( b_2 \) (as a consequence of the propagation of the exact-one constraints).

Finally, by checking the values of the corresponding propositional variables, the \text{value()}\ procedure returns a set of those values which are not excluded from the domain of the given object variable.
4.4 The constraint network

By relying on the sat_core module, as described in Section 4.1, on the linear real arithmetic theory, as described in Section 4.2, and on the object variables theory, as described in Section 4.3, it is possible to create new propositional, numeric and object variables and enforcing constraints on them. Moreover, in case of inconsistencies, it is possible to generate explanations which can be used to allow no-good learning, avoiding the reemergence of similar, inconsistent, situations, and non-chronological backtracking.

Thanks to the linear real arithmetic theory, which abandons arc-consistency propagation in favor of the Simplex method, the introduced constraint network is able to manage numeric variables with an initial infinite domain. Nonetheless, inconsistent situations are quickly detected and exploited, in concert with other theories and with the sat_core module, to allow no-good learning and non-chronological backtracking. Numeric variables can be used to represent, uniformly, different numeric aspects of the planning problem including time, resource consumption amounts, resource capacities, etc. The link between temporal variables and other dimensions becomes, therefore, easy, allowing to relate, for example, resource consumption amounts to temporal spans.

As mentioned earlier, the sat_core module does not implement any search algorithm. Unless it is an explicit consequence of constraints propagation, none of the variables will be assigned. Search is, in other words, demanded to the sat_core’s user applications that can choose which literal to assume and, consequently, what variables consider as relevant in generating solutions. The resulting constraint network will be exploited, in the following chapter, both for generating a new heuristic and for propagating constraints during the resolution process.
By relying on the data structures presented in Chapter 4, this chapter introduces two new heuristics and a new solver architecture for solving timeline-based planning problems. The overall proposed idea consists in applying, in a coarse way, all possible resolvers for all possible flaws until some termination criteria, i.e., unifications and resolvers which do not add further flaws, is met. Specifically, since flaws and resolvers are causally related (i.e., resolvers might introduce flaws which are solved by other resolvers) it is possible to build an AND/OR graph for representing such causal relations. By exploiting the topology of such a graph it is possible to generate an estimation of “how far” a flaw is from being solved and exploit this estimation for guiding the resolution process. Specifically, taking inspiration from the $h_{add}$ and the $h_{max}$ heuristics introduced in Section 2.1.1, the cost of a resolver, which can be seen as an AND node, can be estimated as the sum (or the maximum, in case of $h_{max}$) of the estimated costs of the flaws introduced by the resolver itself plus an intrinsic resolver’s cost, while the estimated cost of a flaw, which can be seen as an OR node, can be estimated as the minimum of the estimated costs of its possible resolvers. It is worth noting that generating such a cost estimation is equivalent to identifying a critical path within the causal graph that can be used for choosing resolvers at resolution time, hence, the name of the heuristic.

Conversely from the $h_{add}$ and the $h_{max}$ heuristics for classical planning, the proposed heuristic does not rely simply on propositions. The state is, indeed, described, also, by numerical variables representing, for example, time, precluding, in the most general case, the possibility of generating ground expressions. This enhancement results into the need to manage the atomic formulas while keeping the variables lifted and, hence, similarly to what has been done in [46] in the case of functional STRIPS [52], strongly relying on constraint propagation capabilities already in the heuristic generation phase. Furthermore, continuing in the wake of [29] and somehow inspired by [111], the concept of reification is applied, in addition to resolvers, to flaws as well. As a consequence, rather than maintaining the flaws into an agenda data structure, as proposed in Section 2.3 and implemented in common timeline-based planners, flaws
CHAPTER 5. THE ORATIO SOLVER

Figure 5.1: The topology of the causal graph is exploited for recognizing a critical path.

arise as a consequence of the sat_core module’s propagation. Additionally, the atom network should be adapted so as to maintain, besides the sole atoms in the solver’s current state, also all the possible atoms that may exist in all possible states that can be reached by the planner. As we shall see, this can be achieved by introducing a state variable $\sigma$ to be assigned to each atom of the atom network and including, among the constraints, also the causal ones. This paradigm shift, however, requires a complete architecture refactoring which has lead to the development of a new planner, called ORATIO, and to the introduction of a new language, called RIDDLE (from oRatIo Do- main Definition Language), for expressing timeline-based planning problems.

Figure 5.1 shows a synthetic example of such a generated graph. In the picture, flaws are represented through boxes and, since any of their resolvers should be applied to resolve a flaw, are OR nodes. As an example, flaw_0 can be solved either by applying the res_0 resolver or by applying the res_1 resolver. Resolvers, on the other hand, are represented through ovals and, since all the flaws introduced by them must be solved, are AND nodes. As an example, resolving flaw_0 through res_0 introduces both the flaw_1 and flaw_2 flaws. The critical path for solving the problem is highlighted in the figure. In particular, once built the graph, a possible solution for the planning problem might consist in applying, at first, resolver res_0. This results in the “activation” of flaws flaw_1 and flaw_2 which might be solved by applying, respectively, the res_3 and res_4 resolvers. It is worth noting that resolvers res_1, res_2, res_5 and res_6, as well as the flaws flaw_3, flaw_4, flaw_5 and flaw_6, although not included into the final solution, are nonetheless considered into the graph. Furthermore, their associated atoms and constraints, included the causal ones, must be introduced into the atom network and, hence, into the underlying constraint network. Building such a graph allows to consider, in a relaxed way, all the possible plans which can be generated starting from a given planning problem. Relaxation, in particular, comes from neglecting the possible interactions among resolvers/subgoals. It might be the case, for example, that resolvers res_3 and res_4, both in the identified critical path, although individually applicable, are not applicable simultaneously.

Before going into further details in explaining the heuristic, however, it is worth to introduce some of the main concepts underlying the solver. In particular, as a consequence of flaws and resolvers reification, it is worth to explain in detail how such data
structures are made. This would allow to have a better idea of how the graph is built and how the solving process works.

5.1 The ORATIO architecture

Figure 5.2 broadly describes the ORATIO architecture. While the lower part refers to the data structures introduced in Chapter 4 (i.e., the sat_core presented in Section 4.1, the LRA theory presented in Section 4.2 and the OV theory presented in Section 4.3), the upper part refers to the main data structures introduced in this chapter. To begin, the external interface of the ORATIO solver, through which a user can specify problems and solve them, is the following.

```cpp
class solver : theory {
    void read(string script)
    void read(vector<string> files)
    void solve()

    sat_core sat
    lra_theory la
    ov_theory set
}
```

Intuitively, the solver is feed through the two read() methods which, respectively, parse a RIDDLE script and a collection of files containing RIDDLE code (for a complete description of the RIDDLE language refer to Chapter 6). The problem is subsequently solved through the solve() method which either finds a solution or returns with an unsolvable exception. Finally, the solver maintains a reference to the sat_core, to the linear real arithmetic theory and to the object variables theory presented in Chapter 4.

It is worth to notice that the solver is, itself, a theory and is bounded to the sat_core. Specifically, as a theory, the solver is notified for the assignment of the bound variables by the sat_core which, additionally, calls the solver’s push() and pop() methods for allowing it to store the context before creating a branch. As will be shown soon, this will allow the solver to receive updates about flaws activation and resolvers’ state.
Figure 5.3 describes, in further details, some of the most important data structures required by the solver for maintaining information. Specifically, \textit{env} and \textit{scope} represent abstract data structures for retrieving instances, types and predicates. The \textit{item} structure is used for representing instances of a specific type. It contains, indeed, a reference to the \textit{type} data structure which is used for representing such types. It is worth to notice the two depicted operations for the \textit{item} type (i.e., \textit{equates} and \textit{eq}). Specifically, \textit{equates} compares efficiently, yet incompletely, two items returning a boolean indicating whether the comparison was successful. This method makes it possible to make the unification checking process more efficient by discarding, without the need to create new variables, those unifications which are trivially unfeasible. Two arithmetic variables whose domains are, respectively, \([0, 10]\) and \([20, 30]\), for example, regardless of the involved constraints, can never be made equal. On the other hand, the \textit{eq} method deeply compares two items returning a propositional variable which, like the reified constraints described in Section 4.1.3, represents whether the unification constraint is satisfied or not. Both the methods are applied iteratively to all the member fields of the items (i.e., those contained in the \textit{items} field of the \textit{env} data structure). It is worth to notice that although knowing two items equate does not imply their unifiability, knowing two items do not equate ensures that the two objects are not unifiable.

\textbf{Bool items, arithmetic items and var items} are specific data structures for managing instances of primitive types which, in addition to keeping references to literals, linear expressions and object variables, as defined in Chapter 4, redefine the \textit{equates} and \textit{eq} operators. Finally, while atoms are a special kind of items, predicates are modeled as a special kind of types. This allows to efficiently recover all atoms having a given predicate and, thus, useful for computing unifications, and, at the same time, to associate a predicate to each atom.

It is worth to notice that while the parameters associated to the atoms are retrievable through the \textit{items} field of the \textit{env} abstract data structure, to which the \textit{equates} and \textit{eq} methods also refer, each atom has its own state variable \(\sigma \in \{\text{INACTIVE, ACTIVE, UNIFIED}\}\). Such a variable indicates the state of the atom and allows for all the possible atoms, including those which are not in the current partial solution, to stay, together, in the same atom network. By exploiting the features of the SMT-based constraint network presented in Section 4, these variables can be represented through propositional variables with the assumption that \textit{UNASSIGNED} variables indicate \textit{INACTIVE} atoms, \textit{TRUE} assignments indicate \textit{ACTIVE} atoms and \textit{FALSE} assignments indicate \textit{UNIFIED} atoms.

The last aspect to consider regards timelines. As already said in Chapter 3, and as it will be further detailed in Section 5.6, timelines are specific \textit{types} whose role consists in generating further flaws which are peculiar to the specific timeline type (e.g., different states overlapping on a state-variable, resource overusages, etc.).

### 5.2 Building the causal graph

As already mentioned in Section 2.4, the key question which pervades the whole thesis is: “since some of the rules have not yet been applied, how is it possible to reason on atoms that have not yet been added in the current atom network?”. The answer pursued
5.2. BUILDING THE CAUSAL GRAPH

by this chapter is, trivially: it is not possible. The main idea pursued to work around this issue is to apply, in parallel, enough resolvers, in a rather coarse way, so as to provide some “insight” on how to solve the planning problem. The result is a Directed Graph (DG), called causal graph, which interleaves flaws and resolvers. By exploiting the topology of such network, flaws and resolvers can than be evaluated through heuristics similar to the $h_{add}$ and $h_{max}$ described in Section 2.1.1, subject to replacement of goals with flaws and actions with resolvers. Notice that the resulting graph is similar to the AND/OR graph generated in [8], yet with a smarter weighting function.

It is worth to notice that different resolvers for a given flaw might interfere each other. In Figure 5.1, as an example, either resolver res$_0$ or resolver res$_1$ is applied for solving flaw flaw$_0$. Applying both resolvers might easily result in introducing inconsistencies. For this reason, in general, resolvers are not added directly within the partial plan but are associated to a propositional variable $\rho$ that determines their activity according to the reification principle. Once built the causal graph, the role of the solver, rather than collecting flaws and applying resolvers, will be “reduced” to the problem of assigning truth values to such variables. Theory propagation will do the rest. Clearly, applying resolvers could introduce new flaws (e.g., subgoals and/or disjunctions) which have to be causally constrained to the generating resolver’s $\rho$ variable. This can be achieved by assigning to each flaw another propositional variable, $\phi$, and constraint it, through a set of clauses called causal constraints, to the $\rho$ variables of the resolvers that have given rise to the flaw.

The construction of the graph and the addition of causal constraints is possible thanks to the presence of the following variables which, together with the sat_core and to the theories, represent the state of the solver.

```
var $v = \top$  // the controlling variable
queue< flaw > flaw_q  // the flaw queue
set< flaw > flaws  // the current active flaws
map< atom, flaw > reason  // the reason for having an atom
stack< layer > trail  // the list of resolvers in chronological order
```

Specifically, the $v$ variable represents the current propositional variable from whose trueness all the assertions depend. The idea behind this variable is that every assertion
α which is asserted as, for example, a linear constraint, is not asserted directly, rather, the \((\neg \nu, \alpha)\) clause is added. The presence of the \(\nu\) variable allows to “execute” RIDDLE snippets in a uniform manner regardless the code represents the problem’s goals and facts, the body of a rule or a disjunct of a disjunction. This is allowed, among other things, by the reorganization of the problem as described in Chapter 3 which makes the problem requirement and the rule requirement homogeneous. The \(\nu\) variable is initialized at \(\top\_var\) since the problem requirement, that is the first RIDDLE snippet that is executed, must be in the solution. From time to time, however, the \(\nu\) variable will be replaced with the \(\rho\) variable of the resolver that is going to be applied.

Similarly to the \texttt{sat_core}'s \texttt{prop_q} variable, the \texttt{flaw_q} variable maintains a queue of flaws for which resolvers have not yet been computed. The \texttt{flaws} variable represents the set of currently active flaws. Since the resolving procedure catches the flaws that are to be solved from this variable, this set represents the counterpart of the agenda data structure as described in Chapter 3. The \texttt{reason} map is intended for keeping track of the flaws that introduced the atoms and is aimed at retrieving, given an atom, its reason flaw. Finally, the \texttt{trail} state variable is responsible for maintaining, in chronological order, the chosen resolvers as well as for storing the context so as to allow backtracking. Specifically, the \texttt{layer} data structure is described by the following code snippet.

```cpp
class layer {
    lit decision
    map<flaw, rational> f_costs
    map<resolver, rational> r_costs
    set<flaw> new_flaws
    set<flaw> solved_flaws
}
```

In particular, \texttt{decision} represents the literal which introduced the new layer (typically, the directed \(\rho\) variable of a resolver). The \texttt{f_costs} (\texttt{r_costs}) map is used for storing, for each flaw (resolver) whose cost has been updated, the cost before the update and will be used, when backtracking, for restoring the costs. The \texttt{new_flaws} and the \texttt{solved_flaws} sets, respectively, are used for storing those flaws which have become active and those flaws which have been solved at the current decision level. Again, when backtracking, these sets will be used, respectively, for removing (those who have been activated) and for re-adding (those who have been solved) flaws from/to the \texttt{flaws} variable. Finally, the solver relies on the following maps for efficiently retrieving flaws and resolvers by their \(\phi\) and the \(\rho\) variables.

```cpp
map<var, vector<flaw>> \(\phi\)s // allows flaw’s retrieval by its \(\phi\) var..
map<var, vector<resolver>> \(\rho\)s // allows resolver’s retrieval by its \(\rho\) var..
```

Notice that, since different flaws and different resolvers might share the same propositional variable, the above two maps have as index the \(\phi\) and \(\rho\) variables while they have as value a collection of, respectively, flaws and resolvers. Before explaining how the graph is built, however, a detailed explanation of flaws and resolvers is required.
5.2. BUILDING THE CAUSAL GRAPH

5.2.1 Representing flaws

As mentioned in Section 2.3, while flaws can be of different types and can arise for different reasons, what they all have in common is that a search choice is necessary to solve each of them. Such choices are represented by means of a collection of resolvers, called the resolvers of the flaw, each of which can resolve the flaw. Additionally, each flaw has another collection of resolvers, called the cause for the flaw, containing all those resolvers that give rise to the flaw. As an example, since top-level flaws (i.e., those introduced directly by the problem instance as goals) have no cause, the cause set, for them, is empty. The cause set of a subgoal contains the sole resolver which introduced the flaw (i.e., the application of a rule). Finally, inconsistencies arising from types (e.g., different atoms temporally overlapping on a state-variable) might have different resolvers as cause (i.e., in case of a state-variable, the resolvers that introduced the temporally overlapping atoms). Furthermore, each flaw has a third collection of resolvers, initially equal to the cause, which contains the supports of the flaw and represents all those resolvers which are supported by the flaw. Unlike the cause, initialized, once and for all, the latter collection is dynamically updated, for example, every time an atom flaw is used as the target of a unification. In summary, the state of a flaw is described by the following state variables.

```
bool exclusive
bool expanded = false
var φ
rational est_cost = +∞ // the estimated cost of the flaw...
vector<resolver> resolvers // the resolvers for the flaw...
vector<resolver> cause // the cause for having the flaw...
vector<resolver> supports // the resolvers supported by the flaw...
```

Specifically, a flaw is called exclusive if any of its resolvers implies the negation of the other resolvers. Consequently, a non-exclusive flaw allows the application of more than one of its resolvers. Since object variables can assume no more than one value, flaws arising as a consequence of object variables are exclusive resolvers. Similarly, tokens associated to goals can either unify or be activated, therefore flaws arising by goals are also exclusive flaws. On the contrary, flaws arising as a consequence of disjunctions, as a consequence of values temporally overlapping on state-variables as well as resources overusages, represent examples of non-exclusive flaw since the different choices can be simultaneously part of a solution. The expanded variable maintains information about whether the resolvers of the flaw have been computed or not. φ is the propositional variable associated to the flaw and is initialized through the init() initialization procedure described later, est_cost is the currently estimated cost, computed by the heuristic, for solving the flaw. Finally, the resolvers, cause and supports variables represent, respectively, the resolvers of the flaw, its cause and the resolvers supported by the flaw, as previously defined.

In addition to these variables, the following procedures define the behavior of each flaw.

```
class flaw {
    void init()
    void expand()
}
```
Specifically, the \textit{init()} procedure is responsible for initializing the $\phi$ variable as the reified conjunction of the cause’s $\rho$ variables, introducing a first example of causal constraint. Formally, $\phi = \bigwedge_{r \in \text{cause}} r \rho$ or, in case the cause of the flaw is empty (i.e., the flaw is introduced directly by the problem as a top-level one) $\phi = \top_{\text{var}}$. In other words, the $\phi$ variable is TRUE if and only if all the $\rho$ variables of the resolver constituting the cause for a flaw are TRUE. The $\phi$ variable is than bound to the sat\_core which will notify the solver whenever a value is assigned to it. In particular, in case the TRUE value is assigned to the $\phi$ variable, the flaw becomes active and is added to the flaws set of the active flaws which have to be resolved. Notice that the $\phi$ variable cannot be initialized at flaw’s initialization phase since it might require the introduction of new reified constraints (which, as described in Chapter 4 requires, in turn, the sat\_core being at root level). As a consequence, although not all flaws are created at root level, all of them must be initialized at root level. The \textit{expand()} method, on the contrary, first calls the compute\_resolvers() which, dependently on the specific flaw, is responsible for filling the resolvers collection with the admissible resolvers, and then enforces a new causal constraint which depends on whether the flaw is exclusive or not, setting, finally, the value of the expanded variable to true. In case the flaw is marked as exclusive, the enforced constraint is $\left( \neg \phi, \bigoplus_{r \in \text{resolvers}} r \rho \right)$, where the $\bigoplus$ symbol is used for representing the exactly-one constraint. In case of a non-exclusive flaw, conversely, the enforced constraint is $\left( \neg \phi, \bigvee_{r \in \text{resolvers}} r \rho \right)$. Specifically, the above causal constraints represent logic implications having the $\phi$ variable as antecedent and, respectively, an “exactly one” and a disjunction of the $\rho$ variables of the resolvers as consequent. Finally, the procedure for computing the resolvers of a flaw is dependent on the specific kind of flaw and will be explained, in detail, later. This specificity, in any case, does not interfere with the procedure for building the graph which can be explained, independently of it, earlier.

Table 5.1 contains a summary of the introduced causal constraints emphasizing the OR nature, either exclusive or inclusive, of the flaws. Specifically, the $\phi$ variable is initialized as the reified conjunction of the $\rho$ variables of the flaw’s cause. Additionally, the $\phi$ variable logically implies, in case of exclusive flaws, the reified exactly-one constraint of the $\rho$ variables of the flaw’s resolvers or, in case of inclusive flaws, the reified disjunction constraint of the $\rho$ variables of the flaw’s resolvers.
5.2. BUILDING THE CAUSAL GRAPH

5.2.2 Representing resolvers

On the other side, with respect to flaws, resolvers represent what is to be done in order for a flaw to be solved. Specifically, the state of a resolver is described by the following state variables.

```
var ρ  // the propositional variable associated to each resolver establishing whether the resolver is active or not.
rational intrinsic_cost  // the intrinsic cost of the resolver...
ratonal est_cost = +∞  // the estimated cost of the resolver...
vector<flaw> preconditions // the preconditions of the resolver...
flaw effect               // the flaw solved by the resolver...
```

The $\rho$ variable is the propositional variable associated to each resolver establishing whether the resolver is active or not. The *intrinsic_cost* value represents the intrinsic cost of the resolver. Different resolvers, indeed, can have different intrinsic costs. Since it is possible to characterize the cost of the whole plan as the sum of these values, for all the active resolvers, this expression can be exploited to manage preferences. In case of disjunctions, indeed, such costs can be defined, through the input language, by the user. The *est_cost* rational represents the current estimated cost, computed by the heuristic, for applying the resolver and is initialized at $+\infty$. The role of the graph building procedure and, more in general, of the heuristic, consists in assigning values to the *est_cost* field of the resolvers. Such values will be used by the solving procedure to efficiently select flaws (i.e., those having the most expensive resolvers among their cheapest ones) and their resolvers (i.e., the cheapest ones). The *preconditions* variable is filled while applying the resolver with those flaws which arise by the application itself. These are the flaws which would become active whenever the resolver’s $\rho$ variable becomes TRUE. Finally, the *effect* variable maintains a reference to the flaw which is solved by the resolver. It is worth noticing that the structure of a resolver resembles that of a classical planning action, as defined in Section 2.1, having a set of preconditions, which require to be fulfilled before the action can be introduced, and a single (positive) effect, hence the idea of using an adaptation of the $h_{add}$ and $h_{max}$ heuristics. Despite the simplification, compared to classical planning actions, of having only a single effect, it is worth remembering that both the preconditions’ and effect’s parameters can be continuous, resulting in a size of the state space which can potentially be infinite.

In addition to the previously defined variables, associated to each resolver there is the *apply()* procedure which is responsible for applying the resolver. For example, in case the resolver represents the application of a rule, applying the resolver means adding subgoals and/or constraints, as defined in the rule, by “executing” the rule’s body. Applying a resolver, however, depends, in general, on the specific nature of the resolver (which, in turn, depends on the specific nature of the flaw solved by the resolver). As already mentioned, the application of a resolver does not impose, directly, constraints, rather, it causally links constraints to the $\rho$ variable of the resolver. In other words, whenever the TRUE value is assigned to the $\rho$ variable, all the constraints of the resolver are enforced. Additionally, assigning the TRUE value to the $\rho$ variable also implies that all the preconditions become active, fulfilling the AND nature of the resolvers. This behavior can be obtained easily, and independently from the resolver, by temporarily replacing, from time to time, the solver’s $\nu$ variable before the application of the resolver. Finally, similarly to the flaws’ case, a *get_cost()* procedure is respon-
sible for computing the cost of a resolver as the sum of its estimated cost \( \text{est\_cost} \) and its intrinsic cost \( \text{intrinsic\_cost} \) values.

### 5.2.3 Building the graph

Once the building blocks of the causal graph (i.e., flaws and resolvers), although in their abstract form, have been presented, it is finally possible to explain the proposed algorithm for building the graph.

```cpp
void build() {  
    while(\( \exists f \in \text{flaws} : f.\text{get\_cost}() == +\infty \)) {  
        if(\( \text{flaw\_q}.\text{empty}() \)) throw unsolvable_exception  
        flaw f = flaw\_q.\text{dequeue}()  
        f.\text{expand}()  
        for(resolver r : f.\text{resolvers}) {  
            var tmp\_v = v  
            v = r.\rho  
            r.\text{apply}()  
            v = tmp\_v  
            if(\( r.\text{preconditions}.\text{empty}() \))  
                set_est_cost(r, 0)  
        }  
        if(!\text{sat}\_\text{check}()) throw unsolvable_exception  
    }  
}
```

Specifically, while exists, among the current active flaws, a flaw having an infinite estimated cost (i.e., the graph has not been able to find a possible resolution for the flaw) and the flaw queue is not yet empty (i.e., it is not possible to find a solution to the problem), a flaw \( f \) is dequeued and expanded. The flaw’s expansion procedure computes all possible resolvers for the \( f \) flaw which can then be applied. Before the application, however, the resolver’s \( \rho \) variable is assigned to the \( v \) variable. Finally, after the application, the \( v \) variable is restored and, in case the resolver’s \( \text{preconditions} \) vector is empty, it is possible to update the cost of the flaw \( f \) as the minimum between the current cost and the intrinsic cost of the resolver. It is worth noting that expanding the flaw initially computes the resolvers and then enforces the causal constraints. During the graph building procedure the \( \text{sat\_core} \) might become inconsistent and is, therefore, checked. In case of failure, it is possible to establish that the problem has no solution, therefore, an exception is launched and the procedure terminates.

Since efficiency is strictly related to the number of involved variables, reducing the number of atoms in the planning graph may result, in some cases, in a significant increase in performance. As a consequence, the graph building procedure can be enhanced for managing deferrable flaws. Specifically, a deferrable flaw is a flaw having any of its ancestors with a finite estimated cost. The idea, here, is to leave out those flaws constituting the preconditions of already, possibly, solved flaws, deferring them in case we find, through the search, that the ancestor was not, actually, solvable. The procedure for checking whether a flaw is deferrable is straightforward and reported for sake of completeness.

```cpp
bool is_deferrable(flaw f) {  
    queue<\text{flaw}> q  
    
```
5.2. BUILDING THE CAUSAL GRAPH

Given the above procedure, it is possible to rewrite the graph building procedure so as to do not expand deferrable flaws, nor to apply its resolvers, re-enqueuing them, on the contrary, for further future checks.

5.2.4 Updating flaws’ and resolvers’ estimated costs

Updating the estimated cost of a resolver might result in updating the estimated cost of other flaws and resolvers depending from the updated one. Potentially, the resolvers’ costs propagation procedure can reach those flaws contained in the flaws variable thus affecting, among other things, the exit condition of the graph building procedure. For this reason, it is necessary to introduce a separate procedure dedicated to propagating the flaws’ and resolvers’ estimated costs.

```java
void set_est_cost(resolver r, rational cost) {
    if (r.est_cost != cost) {
        if (!trail.empty() && !trail.top().costs.contains(r))
            trail.top().r_costs[r] = r.est_cost
        r.est_cost = cost
        resolver bst_res = min_{r:effect resolvers} r.get_cost()

        rational efct_cost = bst_res.get_cost()
        if (r.effect.est_cost != efct_cost) {
            if (!trail.empty() && !trail.top().costs.contains(r))
                trail.top().r_costs[r.eff] = r.effect.est_cost
            r.effect.est_cost = efct_cost
            queue<resolver> resolver_q
            for (resolver c_r : r.effect.supports)
                resolver_q.push(c_r)
        }
    }
}
```

The procedure updates the resolver’s estimated cost, after having stored its previous cost, checking whether the update results in a change in the estimated cost of the resolver’s effect (i.e., the flaw solved by the resolver). If this is the case, all the effect’s supports are enqueued for possible, further, estimated costs updates. The above procedure should, furthermore, clarify the need for distinguishing among flaws’ cause and supports. The \textit{evaluate}() procedure deserves a special mention. Specifically, this procedure returns an estimation of the cost of a set of flaws as the sum (in case the $h_{add}$ heuristic is chosen) or as the maximum (in case the $h_{max}$ heuristic is chosen) of the estimated costs of the flaws.

\subsection{5.2.5 Choosing the \textit{right} flaw and the \textit{right} resolver}

Choosing the right flaw is crucial for performance reasons, however, applying resolvers in parallel results in introducing many flaws which do not require being solved. For this reason, each flaw’s $\phi$ variable is bounded to the \texttt{sat_core} allowing the solver to be notified when a value is assigned to it. Similarly, in case propagation makes any resolver inapplicable, the estimated costs can be updated and propagated so as to improve the heuristic’s accuracy. Each resolver’s $\rho$ variable is bounded to the \texttt{sat_core} allowing the solver to be notified when a value is assigned to such variables. As in any theory, as described in Section 4.1.4, this is accomplished by the \textit{propagate}() procedure which, in case of the solver, is defined as follows.

```cpp
def propagate(lit p, vector<lit> cnf) {
    if(\phi.contains(p.v))
        for(flaw f : \phi[p.v])
            if(p.sign)
                flaws.add(f)
                trail.top().new_flaws.add(f)

    if(ps.contains(p.v))
        for(resolver r : ps[p.v])
            if(!p.sign)
                set_est_cost(r, +\infty)

    return true // cannot fail..
}
```
5.2. BUILDING THE CAUSAL GRAPH

The above procedure allows maintaining updated the flaws solver’s variable with all the active flaws. Additionally, the procedure detects those resolvers which become inapplicable, setting their estimated cost to $+\infty$. It is heuristic’s responsibility, following the $h_{max}$’s intuition, choosing, among such flaws, the most expensive one (i.e., the one that most likely will lead to a conflict, fostering early detection of inconsistencies) and, among its resolvers, the cheapest one (i.e., the one that most likely will lead to a solution). It is worth noting that the overall idea underlying the proposed heuristic consists in extracting the critical path from facts to goals. The cost of the path is given from the intrinsic costs of all the resolvers on which the path passes by. It is worth noticing that the flaws solver’s variable should always contain flaws whose estimated cost is finite, otherwise the solving procedure would not be able to chose among them.

5.2.6 The causal graph applied to the rover domain

In order to further clarify the graph building procedure, it is worth to apply it to the rover domain. Figure 5.4 shows the resulting graph depicting flaws as boxes and resolvers as ovals. The estimated cost is on the top right of each flaw (resolver) while, on their top left, the $\phi$ ($p$) variables. The value of these variables is described by the boxes (ovals) outlines. Specifically, continuous lines correspond to variables whose assignment is TRUE while dashed lines correspond to variables whose current value is UNASSIGNED. In order to do not introduce an overly complex symbolism, in the following we will identify flaws and resolvers by their propositional variables.

Initially, the $flaw_g$ queue contains the flaws (corresponding to variables) $\phi_1$, $\phi_2$ and $\phi_3$. Since these flaws do not have a cause, their $\phi$ variables are initialized at $\top_{var}$. All the three flaws have, initially, an infinite estimated cost, therefore, the graph building procedure dequeues the first one (i.e., $\phi_1$) and expands it. Flaw $\phi_1$, in particular, represents a fact which cannot unify with any other active atom. Its sole possible resolver (i.e., $p_1$) consists, hence, in adding the fact. Additionally, since facts are exclusive flaws, the $\left(\neg \phi_1, \bigoplus_{r \in \phi_1 \text{ resolvers}} r.p\right)$ causal constraint is introduced forcing at TRUE the assignment of $p_1$. Finally, since resolver $p_1$ does not contains any precondition, its estimated cost is set to 0, resulting in an estimated cost of 0 also for the $\phi_1$ flaw. Its least expensive resolver, indeed, is, precisely, $p_1$. The $\phi_2$ flaw is dequeued and solved in a similar way by introducing into the graph the resolver $p_2$. It is now the turn of $\phi_3$ which, not being able to unify, is solved through resolver $p_3$ corresponding to the application of the associated rule and, thus, resulting in the introduction of the two new flaws $\phi_4$ and $\phi_5$. It is worth noting that both the cause collections of such flaws, as well as the supports collections, contain, at this stage, the sole $p_3$ resolver. Additionally, the $p_3$ resolver has two preconditions (i.e., $\phi_4$ and $\phi_5$) and, hence, its estimated cost, as well as $\phi_1$’s one, is still at $+\infty$. Finally, both $\phi_4$ and $\phi_5$ are initialized as the reified conjunction of the flaws’ causes, hence, both $\phi_4$ and $\phi_5$ are initialized at $p_3$ and assume the TRUE value.

At this stage, the flaws set contains flaws $\phi_1$, with an estimated cost of 0, $\phi_2$, with an estimated cost of 0, and $\phi_3$, with an estimated cost of $+\infty$, therefore, the graph building procedure goes on. Flaw $\phi_4$ is dequeued from $flaw_g$ and solved by introducing
the resolver $\rho_4$ which, in turn, introduces the flaw $\phi_6$. Flaw $\phi_5$ is dequeued and solved by introducing the resolver $\rho_5$ which introduces the disjunction flaw $\phi_7$. Flaw $\phi_6$ is dequeued and solved by introducing the resolver $\rho_6$ which introduces the flaw $\phi_8$. Till now, all the causal constraints are such that both $\phi$ and $\rho$ variables must assume the TRUE value and, hence, all the involved constraints are propagated. It is now flaw $\phi_7$’s turn whose expansion introduces the disjunct resolvers $\rho_7$ and $\rho_8$. Both the $\rho_7$ and $\rho_8$ variables assume, conversely to previous ones, the UNASSIGNED value. In other words, it is not clear, at this time, whether $\rho_7$ and $\rho_8$ will assume a TRUE or a FALSE value in the plan. This decision will be taken during the resolution phase, furthermore, their estimated cost is, for both of them, still $+\infty$. Since $\phi_7$ assumes the TRUE value, it is, however, certain that this decision must be eventually taken. It is now flaw $\phi_8$’s turn whose atom, however, can unify with the atom associated to $\phi_1$. Since unification is possible, resolver $\rho_{10}$ is introduced and the estimated costs are propagated. Specifically, $\rho_{10}$’s estimated cost is set at 0 which, added to the intrinsic cost of the unification (i.e., 1), gives a total cost for the resolver of 1. As a consequence, the estimated cost for flaw $\phi_8$ is set at 1. Cost propagation goes on and $\rho_6$’s estimated cost is set at 2 ($1 + 1$) as well as its effect flaw $\phi_6$’s estimated cost at 3 and $\rho_4$’s estimated cost at 3. It is worth noting that $\rho_3$’s estimated cost is still at $+\infty$ since the maximum estimated cost of its preconditions (i.e., $\phi_5$) is still at $+\infty$. Additionally, $\rho_{10}$ is not the only possible resolver for flaw $\phi_8$. Resolver $\rho_9$, representing the application of the rule, is, indeed, also added. Again, choosing between $\rho_9$ and $\rho_{10}$ is a decision which will be taken at resolution time. In this case, however, while the estimated cost for $\rho_9$ is $+\infty$, the estimated cost for $\rho_{10}$ is 1 making the latter option far more enticing. A similar fate belongs to the other flaws in the $\text{flaw}_q$ queue until the $\rho_{13}$ resolver is introduced resulting in a similar propagation of the costs till to the $\phi_3$ flaw obtaining, eventually, the graph of Figure 5.4. Furthermore, since $\phi_3$’s estimated cost is now finite, the graph building procedure is interrupted and the control flow is passed to the resolution procedure which can now exploit the generated estimated costs for choosing the most expensive flaws and their least expensive resolvers.

It is worth noting that the generated graph reasons purely on causal aspects while demanding constraint reasoning to the underlying constraint network. As an example, the $\phi_4$ flaw is a goal which must be achieved, from a causal point of view, before $\phi_3$ one. Nonetheless, the atom associated to the $\phi_4$ flaw, i.e., $\text{At}(1,5)$, takes place over a period of time which includes the time in which the atom associated to the $\phi_3$ flaw takes
place, as required by the rules in Figure 2.9. In other words, \(\text{TakingPic}(1,5,3,7)\) takes place during the rover is \(\text{At}(1,5)\). The temporal relation is guaranteed by the trueness of the \(\rho_3\) variable and, again, it is not evident from the graph.

5.3 The new solving algorithm

Enhanced with the generated estimated costs, the solving algorithm simply proceeds selecting the most expensive flaw \(f\) from the \(\textit{flaws}\) set, selecting its cheapest resolver \(r\) and assuming, through the \textit{sat_core}'s \texttt{assume()} method, the trueness of \(r\)'s \(\rho\) variable.

```c
void solve() {
    build()

    while (true) {
        solve_inconsistencies()

        flaw f_next = select_flaw()
        if (f_next != null) {
            if (f_next.est_cost == +\infty) {
                next()
                continue
            }

            resolver r = f_next.select_resolver()
            if (!sat_core.assume(r.\rho) || !sat_core.check())
                throw unsolvable_exception
            else
                return // a solution has been found!!
        }
    }
}
```

In particular, once built the graph, the procedure enters into the main solving loop. The \texttt{solve_inconsistencies()} procedure solves all the inconsistencies that may arise from the types. The \texttt{select_flaw} procedure purges from the \textit{flaws} set those flaws which are already solved and returns, if any, the most expensive flaw. If the estimated cost for such a flaw is infinite, the \texttt{next()} procedure either backtracks, if possible, or adds a new layer to the graph. Among the applicable flaw’s resolvers, the cheapest one is chosen by the \texttt{select_resolver()} procedure, it’s \(\rho\) variable is assumed as \texttt{TRUE} and the \textit{sat_core}'s \texttt{check} procedure is called. Finally, in case there are no more flaws, the \texttt{select_flaw} procedure returns a \texttt{null} flaw, indicating that a solution to the timeline-based planning problem has been found.

By applying the above algorithm to the graph of Figure 5.4 the most expensive flaw, i.e., \(\phi_7\), is selected and its cheapest resolver, i.e., \(\rho_2\), is chosen. The \texttt{TRUE} value is assigned to variable \(\rho_2\) and the \textit{sat_core}'s \texttt{check} procedure is invoked guaranteeing constraint consistency. As a consequence of constraint propagation, \(\phi_{11}\) flaw becomes active and is added to the \textit{flaws} set. The \texttt{select_flaw} procedure, again, selects the most expensive flaw which is either \(\phi_8\) or \(\phi_{11}\) (they are equally evaluated). Suppose, as an example, that \(\phi_8\) is selected, its cheapest resolver, i.e., \(\rho_{10}\) is selected and its variable is assumed. As a consequence of the unification, the \(?x\) and \(?y\) variables of the atom associated to the \(\phi_8\) flaw, for example, would both assume the value 0 just as the values
of the parameters of the atom associated to fact $\phi_1$. Finally, resolver $\rho_{13}$ would be selected for the sole remaining active flaw $\phi_{11}$ producing, without ever backtracking, a solution for the original planning problem.

5.3.1 The role of pruning

It is worth noting that after the invoking of the graph building procedure many flaws might have been remained in the $f_{law\_q}$ queue which have not yet been expanded. In case of the causal graph of Figure 5.4, for example, $\phi_{10}$, $\phi_{12}$ and $\phi_{13}$ are still in the queue. Two considerations on such flaws are valid: (i) their estimated cost is necessarily $+\infty$ and (ii) it is not known, giving the current graph topology and the computed estimated costs, how to solve them. There is no reason for maintaining the possibility for such flaws to be chosen by the $select\_flaw$ procedure and is, hence, possible to assume the value of their $\phi$ variables as negated. Similar to the one proposed in [111], this kind of pruning is quite efficient since it strongly reduces the size of the search space. In case of Figure 5.4, for example, assigning FALSE to $\phi_{12}$ results in the impossibility of choosing the $\rho_9$ resolver whose $\rho_9$ variable, as a consequence of propagating the causal constraints, becomes FALSE. However, since the $\phi_8$ variable assumes the TRUE value, it is the case that either $\rho_9$ or $\rho_{10}$ must assume the TRUE value and, hence, since $\rho_9$ is now FALSE, TRUE is assigned to the $\rho_{10}$ variable. Similarly, FALSE is assigned to both $\rho_{12}$ and $\rho_8$, resulting in the assignment of the TRUE value to the $\rho_7$ and $\rho_{13}$ variables resulting in the solution of the original planning problem without ever needing to start the search procedure.

Although this pruning procedure does not allows us to always find solutions directly, it has, nonetheless, the capability of enclosing the planner’s search space within a well-defined bounding box. This allows us, among the other things, to exploit the no-good learning and the non-chronological capabilities of the underlying constraint network as described in Chapter 4. It is worth to notice, indeed, that in case of realistic domains, inconsistencies barely arise and, consequently, the conflict analysis procedure is rarely invoked. By enclosing the planner’s search space within precises bounds, conflicts can occur much more easily and a finer analysis can be conducted so as to reduce unfruitful parts of the search.

As already briefly mentioned, a similar pruning is performed in [111] which, however, rather than on pure causality, relies on the plan’s length. Although in case of prefixed durations the pruning is the same, in case of flexible durations it might be hard to foresee the plan’s makespan at heuristic building time resulting in prunings which can be either too strict, requiring a relaxation of the pruning, or too broad, resulting in an excessive search. By intervening on the purely causal aspects while leaving aside temporal reasoning, the proposed approach is more robust on this side.

Unfortunately, it is not always the case that the graph building procedure introduces enough flaws and resolvers for finding a solution and, similar to [111], it might happen that the pruning requires a relaxation. Specifically, it is possible to expand those flaws which remained pending within the $f_{law\_q}$ queue. However, since their $\phi$ variables assume now the FALSE value as a consequence of pruning, expanding the graph might result in a waste of time. Furthermore, all the no-goods learned during the resolution process, might be strictly related to the graph and, hence, to the pruning. A possible
workaround consists in introducing a new propositional variable $\gamma$, representing the validity of the graph, and linking the pruning by means of clauses to $\gamma$. By assigning TRUE to $\gamma$, the pruning would be enforced. In case of failure, however, the generated no-goods would involve the $\gamma$ variable assigning FALSE to it (in which case the graph would be recognized as incomplete/invalid and a graph expansion procedure would be called). In case of the rover domain, as an example, the constraint $(\neg\gamma, \neg\phi_{12})$ would guarantee, after assigning TRUE to $\gamma$, the negation of $\phi_{12}$ and the smooth execution of the search algorithm without the risk, among other things, that the heuristic becomes blind by assigning infinite values to the costs of an active flaw.

5.4 Increasing the heuristic accuracy

As already mentioned, the estimated costs for the nodes of the causal graph are strongly inspired by the $h_{add}$ and $h_{max}$ heuristics for classical planning. Such heuristics, however, as stated by its authors in [10], might be not accurate enough. Specifically, the heuristic completely ignores the possible interactions between the subgoals introduced by the application of the resolvers. While this might not be a problem, in some cases, there are cases in which the interactions are so strong that neglecting them could result in a not so accurate estimation of the costs. Suppose, for example, there is a rule that introduces two goals which, according to some constraints, cannot be both achievable at the same time although, individually, they are. In such a case, the two goals are called mutually exclusive or, more shortly, mutex and, in the classical planning case, are managed, for example, by the $h^m$ class of heuristics [61, 60]. Exception made for some trivial cases in which the constraint network’s propagation capabilities are sufficient to detect such inconsistencies, an estimated cost would be associated to each of the goal and their sum (or, according to the chosen heuristic, maximum) of them is associated to the resolver. Since the subgoals are not achievable at the same time, however, it would be not possible to apply the resolver. In other words, its estimated cost should be $+\infty$ and the FALSE value should be assigned to its $\rho$ variable. This would allow, at heuristic building phase, to produce a more accurate estimate of all the goals and resolvers in the ancestors of the inapplicable resolver. Although more accurate, however, it has not yet been found an efficient graph building procedure which would build such a heuristic estimate without performing too much constraint propagations which, currently, would result to be excessively expensive.

5.5 Different flaws (and their resolvers)

This section presents, more in detail, the basic flaws and their resolvers which constitute the backbone of the environment. Specifically, the presented flaws are introduced for managing object variables, allowing to choose a value for them, disjunctions, allowing to choose for a disjunct, and atoms, which might be either activated resulting, in case of goals, in the associated rule’s application, or unified with any of the other already activated atom.
5.5.1 Object variable and disjunction flaws

The first and, probably, easier flaw, is the object variable flaw. Similarly to CSPs’ variables, whenever a new object variable, having different allowed values in its domain, is introduced (e.g., as a parameter of an atom), a value for such a variable has eventually to be chosen. This decision can be easily integrated into the resolution process by exploiting the flaw and the resolver concepts. Specifically, an object variable flaw is created for each object variable maintaining a reference to the variable itself. The implementation is straightforward and will not be detailed. The `compute_resolvers()` procedure simply creates a resolver for each of the allowed values establishing for them an intrinsic cost of $1/#\text{values}$ (with $#\text{values}$ representing the number of allowed values). Assigning the `TRUE` value to the $\rho$ variable of such resolvers would result in assigning a value to the associated object variable. Since each object variable has a propositional variable for each of the allowed values, as described in section 4.3, such a variable can be used in place of the $\rho$ variable. Clearly, object variable flaws are exclusive. Notice that the cost assignment is uniform among the flaw’s resolvers and, although it allows choosing those object variables which have less allowed values first (i.e., those that have a higher cost), it does not help in choosing a value for the variable. Finally, it is worth recalling that the atoms’ $\tau$ variables are, to all effects, variables and, as such, could introduce object variable flaws.

Unlike the object variable flaws, disjunction flaws are aimed at managing disjunctions. Whenever a new disjunction arises, indeed, a disjunction flaw is created. Even in this case, the implementation is straightforward and will not be detailed. The `compute_resolvers()` procedure, indeed, simply creates a resolver for each of the disjuncts of the disjunction. The intrinsic cost of such resolvers can be either specified by the user, so as to realize a strategy for preferences management, or it can be defaulted to 1. The `apply()` procedure for the disjunct resolvers is responsible for “executing” the RIDDLE code inside the disjunct after having temporally replaced the $\nu$ variable of the solver with the corresponding $\rho$ variable. This would guarantee that assigning the `TRUE` value to the $\rho$ variable of such resolvers would result in activating the corresponding disjunct. Contrary to the object variable flaws, disjunction flaws are non exclusive.

5.5.2 The atom flaw

Among the different basic flaws, the atom flaw, for managing facts and goals, is definitely the most complex and interesting one. Specifically, atom flaws are aimed at justifying atoms (i.e., either by activating them or by unifying them with other already active atoms), representing either facts or goals. It is worth to recall that facts can unify just as goals. Activating facts, however, does not imply applying the corresponding rule. The `compute_resolvers()` procedure is mainly aimed at finding adequate candidates for unification creating, for each of them, a unification resolver. It is worth to notice, however, that each unification resolver has, as a precondition, the flaw associated to the target of the unification (i.e., the flaw associated to the atom with which unification is going to happen). For this reason, one must be careful not to introduce unification resolvers which could result in causal cycles into the causal graph. In ad-
5.5. DIFFERENT FLAWS (AND THEIR RESOLVERS)

dition to the unification resolvers, \texttt{compute_resolvers()} always adds a further resolver aimed at applying the corresponding rule, in case the atom is a goal, or at simply activating the atom, in case it is a fact. The following code snippet describes the atom flaw’s \texttt{compute_resolvers()} procedure.

```plaintext
void compute_resolvers () {
    set<flaw> ancestors
    queue<flaw> q
    q.add(this)
    while (!q.empty()) {
        flaw f = q.dequeue()
        if (!ancestors.contains(f)) {
            ancestors.add(f)
            for (resolver s : f.cause)
                if (sat.value(s.p) != False)
                    q.add(s.effect)
        }
    }
    for (atom a : atm.tp.get_instances()) {
        if (a == atm) continue
        if (sat.value(atm.σ) == False) continue
        if (!atm.equates(a)) continue
        atom_flaw trg = reason[a]
        if (!trg.expanded) continue
        if (ancestors.contains(trg)) continue
        var eq_v = atm.eq(a)
        if (sat.value(eq_v) == False) continue
        vector<lit> unif_lits = {atm.σ, ¬a.σ, eq_v}
        unify_atom u_res = new unify_atom(atm, a, unif_lits)
        resolvers.add(u_res)
        u_res.preconditions.add(trg)
        trg.supports.add(u_res)
        set_est_cost(u_res, trg.get_cost())
        sat_core.add_clause(trg.φ ∨ ¬u_res.p)
        ps.add(u_res.p)
        bind(u_res.p)
    }
    if (is_fact) resolvers.add(new activate_fact(atm))
    else resolvers.add(new activate_goal(atm))
}
```

The first section of the above procedure is aimed at computing the ancestors of the atom flaw so as to avoid causal cycles. Next, all the atoms having the same predicate are checked for unification, however, many of them can be easily discarded. First of all, atoms cannot unify with themselves nor with other unified atoms. The \texttt{equates()} procedure is thus invoked for efficiently excluding also those atoms which are obviously too different for allowing unification. The atom flaw associated to the target atom is than retrieved through the solver’s \texttt{reason} map (which can be filled, for example, at atom flaw initialization phase). In order for the unification to be valid, such a flaw must have already been expanded and should not be part of the flaw’s ancestors so as
to avoid the introduction of cycles within the causal graph. Finally the `eq()` procedure is invoked returning the propositional variable representing the reified conjunction of all the equality constraints of the parameters pairwise. If this variable already assumes the `FALSE` value, the target atom is excluded from the unification, otherwise, we can establish, with reasonable certainty, that the unification will be successful and, hence, a unification resolver is added to the collection of resolvers. Additionally, the flaw associated to the target atom is then added to the preconditions of the unification resolver and the unification resolver is added to the supports (yet, not on the cause) of the flaw associated to the target atom. It is worth noting that this is the only place in which the `supports` set is differentiated from the `cause` one. The cost of the unification resolver is set as the cost of the target atom flaw. Furthermore, a clause that invalidates the unification resolver in case the target flaw becomes inactive is also added. Finally, the unification resolver and its $\rho$ variable are added to the $\rho$s map and the $\rho$ variable is bound to the `sat_core` waiting for notifications which might result in resolvers estimated costs updates.

The unification resolver, the activate fact resolver and the activate goal one have, respectively, intrinsic cost of 1, 0 and 1. Additionally, for sake of completeness, it is worth to describe in detail the three `apply()` procedure of the involved resolvers. Specifically, the unification resolver’s `apply()` procedure simply enforces that all literals of the unification literals (i.e., the target atom’s $\sigma$, the negation of the unifying atom’s $\sigma$ and the equality propositional variable) list must be true whenever the unification resolver’s $\rho$ variable becomes true.

```c
void apply() {
    for (lit l : unif_lits)
        sat_core.new_clause(¬ρ ∨ l)
}
```

In case of the `apply()` procedure of the resolver demanded at activating facts, it simply enforces that the atom’s $\sigma$ variable is `TRUE` whenever the resolver’s $\rho$ variable becomes true.

```c
void apply() {
    sat_core.new_clause(¬ρ ∨ atm.σ)
}
```

Finally, in case of the `apply()` procedure of the resolver demanded at activating goals, it first enforces that the atom’s $\sigma$ variable assumes value `TRUE` whenever the resolver’s $\rho$ variable becomes `TRUE` and than applies the rule associated to the atom’s predicate.

```c
void apply() {
    sat_core.new_clause(¬ρ ∨ atm.σ)
    atm.tp.apply_rule(atm)
}
```

It is worth to recall that since the $\rho$ variable has been temporarily assigned to the $\nu$ variable, as described in Section 5.2.3, this procedure easily binds the activation of all the constraints and the preconditions contained in the rule to the resolver’s $\rho$ variable.

Notice that atom flaws might introduce cycles in the causal graph which should be managed by the cost propagation procedure. Suppose, for example, in the rover...
domain, that the rover is initially at position (0, 0). Two goals require that the rover is, in any order, at position (0, 1) and (1, 0). The graph building procedure would build the graph in Figure 5.5. In particular, both flaw $\phi_6$ can be solved with resolver $\rho_8$ and flaw $\phi_7$ can be solved with resolver $\rho_{10}$. Removing one of such resolvers would eliminate the cycle while forcing an ordering between the high level goals and thus dropping the completeness of the resolution algorithm. As an example, removing $\rho_{10}$ would force the rover to go at (1, 0) strictly before going at (0, 1).

5.6 Defining the timelines

Since most of the flaws and resolvers are managed by the solver itself, independently from the involved timelines, thanks to the previously introduced data structures (i.e., object variable flaws, disjunction flaws and atom flaws), from a practical point of view, the role of timelines resides, mainly, in generating further flaws for the causal graph which are peculiar to the specific timeline type (e.g., different states overlapping on a state-variable, resource overusages, etc.). From an implementation point of view, therefore, timelines are specific types with a characteristic procedure for generating flaws as described by the following snippet.

```cpp
class timeline : type {
    vector<flaw> get_flaws()
}
```

The implementation of the `get_flaws()` procedure, as well as the kind of returned flaws, is, however, dependent on the specific timeline. The following sections, therefore, detail some of the most used timeline types.

5.6.1 The state-variable

The state-variable type is used for generating state-variable flaws. Since state-variable flaws represent different values overlapping on the same state-variable, resolvers represent, mostly, ordering constraints between atom pairs. The detection procedure for
state-variable flaws recalls the one described in [18] although the lower bounds of the
temporal variables are replaced by the current values of the numeric variables as de-
dscribed in Section 4.2. The procedure, for each state-variable, groups all the active
atoms (i.e., those whose $\sigma$ variable assumes value TRUE) by the value of their $\tau$
variable. In other words, all the atoms of all the state-variables are partitioned according
to the state-variable on which the atoms could apply. Subsequently, for each state-
variable $sv$, the atoms applied on it are grouped by means of their starting and ending
times, according to the following code snippet.

```cpp
// for each pulse, the atoms starting at that pulse..
map<inf_rational, vector<atom>> starting_atoms;
// for each pulse, the atoms ending at that pulse..
map<inf_rational, vector<atom>> ending_atoms;
// all the pulses of the timeline..
set<inf_rational> pulses;
for (atm : atoms) {
    starting_atoms[la.value(atm.items["start"].l)].add(atm)
    starting_atoms[la.value(atm.items["end"].l)].add(atm)
    pulses.add(la.value(atm.items["start"].l))
    pulses.add(la.value(atm.items["end"].l))
}
```

Since the `pulses` set, containing all the starting and ending times, must be ordered,
it is implemented by means of a red-black tree relying on the comparison operators
for the rationals with infinitesimals as defined in Section 4.2. The `pulses` set is than
visited and, relying on the above `starting_atoms` and the `ending_atoms` variables, a
set of currently overlapping atoms is kept updated and, in case its size exceeds one, a
state-variable flaw is created. The procedure is described through the following code
snippet.

```cpp
set<atom> overlapping_atoms
for (p : pulses) {
    overlapping_atoms.add(starting_atoms[p])
    overlapping_atoms.remove(ending_atoms[p])
    if (overlapping_atoms.size() > 1)
        flaws.add(new sv_flaw(overlapping_atoms))
}
```

It is worth noting that the cause for each state-variable flaw is given by the collec-
tion of resolvers that activate the overlapping atoms. As a consequence, a state-variable
flaw will become active whenever all the overlapping atoms, constituting the flaw, be-
come active. State-variable flaws can be solved either by ordering each couple of the
overlapping atoms, let us say $(atm_0, atm_1)$, by means of the $atm_0.end \leq atm_1.start$
constraint or of the $atm_1.end \leq atm_0.start$ constraint. Additionally, state-variable
flaws can be solved by “moving” the atoms on other state-variables by means of the
$atm_0.\tau \neq atm_1.\tau$ constraint. While expanding each state-variable flaw, a resolver is
created for each of the available possibilities.
5.6.2 The reusable resource

Similarly to the state-variable case, the reusable resource type is used for generating reusable resource flaws. Reusable resource flaws represent resource overuses characterized by a concurrent, excessive, use of the resource. Reusable resource flaws are detected in a way which is similar to the state-variable case. However, instead of relying on the number of overlapping atoms, the extraction procedure compares the resource capacity with the concurrent resource usages as in the following code snippet.

```plaintext
set<atom> overlapping_atoms

for (p : pulses) {
  overlapping_atoms.add(starting_atoms[p])
  overlapping_atoms.remove(ending_atoms[p])
}

inf_rational usage
for (a : overlapping_atoms)
  usage += a.items["amount"] .1

if (usage > la.value(rr.items["capacity"]).1))
  flaws.add(new rr_flaw(overlapping_atoms))
```

Considerations related to the flaw’s cause and to the alternative ways for resolving it are analogous to those of the state-variable case. Specifically, the cause for each reusable resource flaw is given by the collection of resolvers that activate the overlapping atoms. Indeed, a reusable resource flaw will become active whenever all the overlapping atoms, constituting the flaw, are active. Similarly to state-variable flaws, reusable resource flaws can be solved either by ordering a couple of the overlapping atoms, let us say \( \langle atm_0, atm_1 \rangle \), by means of the \( atm_0.end \leq atm_1.start \) constraint or of the \( atm_1.end \leq atm_0.start \) constraint, or by “moving” the atoms on other reusable resources by means of the \( atm_0.\tau \neq atm_1.\tau \) constraint. Additionally, reusable resource flaws can be solved by constraining the sum of the overlapping atoms to be less or equal than the capacity of the resource. While expanding each reusable resource flaw, a resolver is created for each of the available possibilities.

5.7 Experimental results

Evaluating the proposed heuristic is not an easy task. Directly comparing a timeline-based planner with a classical planner on a classical planning domain is, indeed, clearly unfair. The expressiveness of the timeline-based approach proceeds along with complex data structures which inevitably results in a slow down of the resolution process. Additionally, the lack of feasibility in making the atoms ground, due to the presence of numeric variables representing, for example, time, introduces the need for propagating constraints which, if performed too many times, results in a source of inefficiency. Classical planning benchmarks do not suffer from such issues. Even those domain models which explicitly represent time, they do it in such a way that temporal reasoning can be mostly ignored, reducing the temporal planning problem to a classical one which, once solved, is re-enriched with temporal aspects. In other words, except for
some sporadic exception, planning benchmarks are not temporally expressive (refer to [28] for a convenient classification of domain models). It is worth to notice, however, that the simple heuristics implemented in iLoC, proposed in Section 3.2, were sufficient to solve temporally expressive problems, yet failed in solving those problems which required a more pervasive use of purely causal reasoning (e.g., the blocks world). The proposed heuristics, indeed, proceed in this direction: without neglecting aspects related to temporal reasoning, the heuristics aim at enhancing those aspects which are mostly related to causal reasoning representing the main cause of inefficiency of timeline-based planners. The first comparisons, therefore, will be made with the old heuristics on those domains already described in Section 3.3.

A C++ implementation of the ORATIO solver, together with benchmarking domains, can be found on-line\(^1\). In order to further investigate the graph building procedure, atom flaws have been implemented in two different ways: (i) the one defined in section 5.5.2 and (ii) a more “accurate” one, in which unifications are further checked. In the latter case, a collection of unification literals is built containing, both for the current flaw and for the target flaw, all the \(\rho\) variables of each resolver from the flaw to the first resolver of the cause which is necessarily active and, together with the target atom’s state, the negation of the unifying atom’s state and the equality variable, is checked to the \texttt{sat\_core}. Only in case all of these literals can be simultaneously made true we can establish, with reasonable certainty, that the unification will be successful and, hence, a unification resolver is added to the collection of resolvers. Additionally, we are interested in understanding how useful is filtering out deferrable flaws in the graph building procedure. As a consequence, four possible solvers have been compared: (a) ORATIO, in which deferrable flaws are considered and unifications are not checked; (b) ORATIO accurate, in which deferrable flaws are considered and unifications are checked through the \texttt{sat\_core} before adding the corresponding resolvers to the atom flaws; (c) ORATIO undef, in which deferrable flaws are not considered and unifications are not checked; and (d) ORATIO undef+accurate, in which deferrable flaws are not considered while unifications are checked. For all of them, the maximum cost of the preconditions, hence following the \(h_{\text{max}}\) approach, was used for estimating costs\(^2\).

5.7.1 Results

Figure 5.6 shows the execution times of ORATIO in case of the Blocks World domain. The first lesson to learn regards the accuracy, intended as checking the unifications by propagating constraints, of the heuristic. A more accurate heuristic, indeed, results in a smaller graph or, more likely, in a graph having less edges, with more precise estimated costs. Since this kind of accuracy requires propagating further constraints, however, it turns out that it does not worth the effort. As shown in Figure 5.7, related to the instance with 22 blocks of the blocks world domain, the number of flaws created by the graph building procedure is the same in the less accurate case (Figure 5.7(a)) and in the accurate one (Figure 5.7(b)). The number of solved flaws in the less accurate case

\(^1\)https://github.com/pstlab/oRatio
\(^2\)Since estimating costs using the sum of the preconditions, hence following the \(h_{\text{add}}\) approach, gave similar results, they are not reported
Figure 5.6: Execution time of different ORATIO implementations on the tower domain.

(Figure 5.7(c)), however, is greater than the number of solved flaws in case the more accurate heuristic is used (Figure 5.7(d)). This can be explained through the fact that, being the estimated costs less accurate, the search algorithm makes more mistakes and, hence, more search is needed. Furthermore, this is confirmed by the overall solving time which is 79% demanded to search in the inaccurate case (Figure 5.7(e)) vs only 42% in the accurate case (Figure 5.7(f)). As shown in Figure 5.6, however, the greater complexity required by a more accurate heuristic does not compensate the reduction of the search time. In addition, as expected, efficiency is strongly affected by the number of involved variables, hence, it benefits from the filtering out of the deferrable flaws in the graph building procedure.

It is worth to notice that both in the less accurate case and in the more accurate one, the number of solved flaws is quite lower than the number of flaws created by the graph building procedure. The reason is twofold: (a) the heuristic, both in the less accurate case and in the accurate one, actually allows to identify the flaws to be solved and the resolvers to solve them; and (b) the pruning procedure, described in Section 5.3.1, is effective in constraining the problem so as to force the resolution of some of the flaws. The latter case is, indeed, the reason for the elimination of all the object variable and the disjunction flaws.

Figure 5.8 compares the critical path heuristic with the previous heuristics (i.e., ALLREACHABLE and MINREACH) and with other classical planners as described in Chapter 3. The figure shows that ORATIO effectively improves over the previous heuristics, suggesting that the taken direction is the right one. Additionally, although classical solvers are still much more efficient than ORATIO which, on classic problems, at present time, is not able to compete, it is worth to notice that ORATIO scales much more gracefully thanks, mostly, to the pruning phase. The reasons for the better efficiency of classical solvers can be sought in the fact that classical planners, typically, rely on a grounding phase of the predicates. Such a grounding, however, is, in general, not allowed in timeline-based planning due to the presence of numeric parameters (e.g., time and/or resource usages). Furthermore, classical planners, rather than explicitly maintaining a representation of temporal variables, reason in terms of subsequent
Figure 5.7: Blocks world solving statistics. Percentage of created flaws (a) in the inaccurate case and in (b) the accurate one. Percentage of solved flaws (c) in the inaccurate case and in (d) the accurate one. Percentage of execution time (e) in the inaccurate case and in (f) the accurate one.
steps, completely eliminating the need of propagating constraints and for managing threats. The presence of numerical variables, overall, if on the one hand requires more execution time, on the other, allows a greater accuracy of the modeled processes. It will be left to the end user’s responsibility to decide for the right balance between expressiveness and performance according to the specific needs of the faced planning problem, while taking into account the fact that performance improved significantly, thanks to the proposed critical-path heuristic, compared to the first timeline-based planners.

A separate discussion must be made regarding the quality of the produced solutions. The critical path heuristic, indeed, produces plans which are optimal from a purely causal point of view which, in the specific case of the blocks world domain, correspond to the optimal solution from a plan length point of view. As a consequence, while ORATIO produces, on these instances, optimal solutions, both OPTIC and COLIN introduce, in the plan, some unstack actions resulting in longer plans.

As regards the Cooking Carbonara domain, Figure 5.9 shows a comparison of the execution times of ORATIO with the previous ALLREACHABLE and MINREACH heuristics as well as with other classical planners. The introduction of the critical path heuristic, as shown in the figure, introduces a slowdown in the overall resolution procedure. Nonetheless, performances remain better than those of the other classical planners, despite, it is worth recalling, such planners solve a slightly easier problem in which the constraint that eating should not start too late after the meal is cooked is removed due to the difficulty of modeling it.

An interesting aspect of this domain, as shown from the charts in Figure 5.10, regards the number of inconsistencies compared to the total number of flaws. Such inconsistencies, in order to be introduced by the graph building procedure, require the introduction of causal constraints and, hence, the need of backtracking at root level. This, at present time, constitutes an element of inefficiency which is not yet completely clear how to solve. If on the one hand it is possible to efficiently manage disjunctions, through the constraint network, it is not yet clear how to introduce new constraints without backtracking at root level.

As regards the Temporal Machine Shop domain, as shown in Figure 5.11, com-
CHAPTER 5. THE ORATIO SOLVER

Figure 5.9: Execution time of different solvers on the cooking carbonara domain.

Figure 5.10: Cooking carbonara solving statistics. (a) Percentage of created flaw. (b) Percentage of solved flaws. (c) Percentage of execution time.
5.7. EXPERIMENTAL RESULTS

Figure 5.11: Execution time of different solvers on the temporal machine shop domain.

pared with the ALLREACHABLE and MINREACH heuristics, performance, unfortunately, rapidly degrade. The reason for such a degradation, on such a domain, consists in the need, for the planner, to extend the graph several times. Specifically, the domain “confuses” the heuristic making the graph building procedure believe that a solution has been found. As a consequence, the searching phase starts for, quickly, failing. A further layer is added to the graph and search starts again until a solution is found. This phenomenon would be highly limited by the introduction of an $h^n$ heuristic which, however, would require too much propagation resulting in an excessively expensive graph building procedure. Hints for workarounds could be found in approaches like those described in [59, 68, 46]. Such approaches, however, have not yet been investigated in the case of timeline-based planning.

Finally, it is worth to spend some words for what concerns the GOAC [49, 15] domain. Specifically, the Goal Oriented Autonomous Controller (GOAC) was an ESA initiative aimed at defining a new generation of software autonomous controllers to support increasing levels of autonomy for robotic task achievement. In particular, the GOAC domain emerged from the GOAC project and is an extension of what we have called the rover domain. Specifically, the domain aims at controlling a rover to take a set of pictures, store them on board and dump the pictures when a given communication channel was available. The interesting aspect of this domain is that communication can only take place within specific visibility windows that take into account the astronomical motions of the planets/satellites which, in some cases, may stand between the transmitting and receiving stations. The presence of these visibility windows, in particular, requires an explicit modeling of temporal aspects in order to adequately plan the transmission of information and can hence easily modeled through and solved by timeline-based planners. The problem is made more interesting by the presence of constraints which include the available resources (e.g., memory and battery) as well as by having a distance matrix, among the possible locations, which might be not completely connected.

Figure 5.12 shows the execution times of different solvers (AP2 [49], EPSL [19] and J-TRE [29], one of the precursors of ORATIO) in solving different instances of
the GOAC problem. In particular, problems are obtained by varying the problem complexity along the the number of pictures to be taken and the number of communication windows (from 1 to 4 visibility windows). Notice that if on the one hand the increasing number of communication windows raises the complexity of the planning problem with a combinatorial effect, on the other hand an higher number of such windows might allow the planner to more easily find room for transmitting. More in general, among all the generated problem instances, the ones with higher number of required pictures and higher number of visibility windows result as the hardest ones. The right mix of causal and temporal aspects makes the GOAC problem particularly complex to the point that the other planners, beyond a certain number of pictures to collect and dump, show serious scalability issues. As shown in the figure, besides being considerably more efficient, ORATIO is also able to solve more complex instances.

In conclusion, the introduction of the critical path heuristics to the timeline-based case proves able to reduce inefficiency issues in those problems which have a prevalence of goal and/or disjunction flaws. In those cases in which most of the flaws are inconsistencies, the ORATIO architecture shows some issues related to the dynamic introduction of new constraints due to the dynamic detection of new flaws making an architecture similar to iLOC more efficient. Finally, the current critical path heuristics
result efficient in those cases in which subgoals have a high degree of independence. In those problems, indeed, in which subgoals interact in a rather complex ways, such interactions are overlooked by the underlying $h_{add}$ and $h_{max}$ heuristics producing inaccurate cost estimations and, even worse, causal graphs which do not contain solutions, resulting in an inefficient alternation of procedures for extending the graph with procedures aimed at searching for a solution.

It should be noted, however, that the proposed heuristics are the first one, as far as I know, which breaks down purely causal aspects from the temporal ones. In addition, similar to the classical planning case, one of the proposed heuristic (i.e., $h_{max}$) is admissible, producing plans which are optimal according to the intrinsic cost of the those resolvers which are active in the final solution. As it will be shown in Chapter 7, these characteristics make the system suitable for integrating into planning other forms of reasoning.
In order to feed timeline-based planners and, more specifically, the ORATIO planner, it is necessary to establish a language for the definition of the problems that is as adequate as possible to carry out its task and, at the same time, easy to use by the end users. This chapter describes the RIDDLE (for oRatIo Domain Definition LanguagE) domain description language which is used by ORATIO for representing physical domains (a complete Extended Backus-Naur form is given in Appendix A). Compared to earlier languages for timeline-based planning (e.g., [16]), the RIDDLE language introduces a pure object-oriented approach to the definition of timeline-based domains and problem definitions and, therefore, allows an higher decomposition of the domain model and an increase of modularity, resulting in a reduction of the the overall complexity at design phase. Furthermore, thanks to the object-oriented approach, UML modeling features can be naturally exploited to enhance the design phase. In addition, aspects related to first order logic are further made explicit, allowing a uniform representation of planning and scheduling concepts. Finally, although the language is based on a multi-sorted first order logic core, from which the object-oriented approach comes, it has been designed for allowing extensibility and is, hence, agnostic of complex types such as state-variables or resources.

Overall, the language is structured so that input problems can be decomposed into different compilation units (i.e., several files) which can possibly interact each other. Each compilation unit can contain, in general, several declarations of different types and/or several statements. Such units are given to the solver in different sorted groups (i.e., a list of lists of files). Furthermore, these groups can be sent to solver at different times so as to provide plan adaptation features. Although not strictly required, it is common practice to separate the type declarations from the statements in different units (e.g., a first unit for type declarations and a second unit for statements, so as to resemble classical planning problems). Furthermore, type declarations can be spread on different units so as to improve model decomposition. In this regard, we tried to facilitate the definition of the domains by internally implementing forward declaration. Specifically, types, methods and predicates can be used before of being declared, under
the obvious assumption that these types (methods and predicates) are defined sooner or later within the same group of units within which the type (method or predicate) is used (or, alternatively, in a group previously sent to the solver). On the contrary, both for defining problem instances and for defining rules’ body, statements are always executed sequentially. It is, therefore, not allowed, for example, to use a variable which has not been previously declared.

To sum up, the suggested methodology, as summarized in the example of Figure 6.1, consists in providing to the solver a set of compilation units containing the definition of types, methods and predicates, so as to inform the solver of the domain model within which it will have to reason. At a later time, a new compilation unit is provided to the solver containing the statements relative to the declaration of the instances, the facts and the goals. At this point, if a solution to the proposed problem exists, the solver will be able to find it and will return $\top$, if not, it will return $\bot$. Finally, if needed, the solution can be adapted, several times, by providing further compilation units.

6.1 An Object-Oriented Language

As introduced in previous chapters, the basic core of the architecture provides an object-oriented virtual environment for the definition of objects and constraints among them. Every object in the environment is an instance of a specific type. There is, as will be further clear soon, an important distinction between primitive types (i.e., bools, ints, reals, enums and strings) and user defined complex types (e.g., rovers, trucks, locations, etc.). Before going further, however, it is worth to introduce some naming conventions.

6.1.1 Identifiers

The names of variables, constants, methods, predicates, as well as types and objects, are called identifiers. A valid identifier for our domain description language is a sequence of one or more letters, digits, or underscore characters (\_). Spaces, punctuation marks, and symbols cannot be part of an identifier. In addition, identifiers shall always begin either with a letter or with an underline character (\_).

The domain description language uses a number of keywords to identify operations and data descriptions; therefore, identifiers created by a domain modeler cannot match
these keywords. The standard reserved keywords that cannot be used as identifiers are:

```
bool, class, enum, fact, false, goal, new, or, predicate, real, return, string, this, true, typedef, void
```

It is worth to notice that the domain description language is a "case sensitive" language. This means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. For example, the variable names MAX and max will be considered as separate identifiers. Here are some examples of identifiers:

```
MAX
max
_first_name
_second_name
```

### 6.1.2 Primitive types

The domain description language is a strongly-typed language therefore it requires every variable to be declared with its type before its first use. The resolving framework needs to have precise information about the type of the variable we want to define. If we want to represent a number, for example, the framework needs to know that the declared variable represents a number and, furthermore, needs to know the specific type of the number (i.e., either an integer or a real).

The simplest way for declaring and instantiating a variable is through the syntax `<type> <id>` that declares a variable of type `<type>` and identifier `<id>`. If declaring more than one variable of the same type, they can all be declared in a single statement by separating their identifiers with commas. Once declared, the variables can be used within the rest of their scope in the program. Unless explicitly specified, the variable will assume a default initial domain which is based on the type of the variable. It is worth to note that, unlike an ordinary programming language, like Java or C++, rather than assuming a value, variables assume a domain of values, therefore the semantic is similar to the variables of a constraint network.

The domain description language provides a set of primitive types that allow us to define basic types of variables. Primitive types for our domain description language are: `bool`, `int`, `real`, `string`, `typedef` and `enum`.

**bool**  The boolean type is the simplest type provided by the domain description language. Booleans are used to represent boolean states (i.e., `true` and `false`). Unless explicitly specified, a boolean variable will assume possible values within the set `{true, false}`. This means that the allowed values of the variable is decided by the solver according to the current constraints. For example, in the limit case in which no constraint insists on the variable, the domain of the variable will be maintained equal to the set `{true, false}`.

**int**  The integer type is used to represent the set of integers, so to speak "without comma", positive and negative (e.g., 1, 2, 43, -89, 4324). The internal representa-
The representation format of integers (i.e., 16 bits, 32 bits, arbitrary-precision, etc.) is dependent on the implementation of the framework and is beyond the description of the language. Unless explicitly specified, an integer variable will assume possible values within the bounds \([-\infty, +\infty]\). Similar to the boolean variables, the allowed values of integer variables is decided by the solver according to the current constraints.

**real**  The real type is used to represent the set of reals, so to speak “with comma”, positive and negative (e.g., 2.7, -3.14, 15.3). Similarly to the integers, the internal representation format of reals (i.e., 16 bits, 32 bits, arbitrary-precision, etc.) is dependent on the implementation of the framework and is beyond the description of the language. Unless explicitly specified, a real variable will assume possible values within the bounds \([-\infty, +\infty]\). This means, again, that the allowed values of the variable is decided by the solver according to the current constraints.

**string**  In order to represent texts, the domain description language provides the string type. Unless explicitly specified, a string variable will assume the empty string value (i.e., "").

**typedef**  The purpose of typedef is to assign alternative names to existing primitive types and possibly to redefine them. This allows us, for example, to define a primitive type called “Angle” which might be a real whose allowed values are within the bounds \([0, 360]\). In general, typedefs are utility constructs that allow the definition of more synthetic code. Indeed, the same behavior can be achieved by defining primitive type variables and imposing constraint on it.

**enum**  When defining an enumerated type variable, it is assigned a set of constants called enumeration set. The variable can assume any of the constants of the enumeration set. Unless explicitly specified, an enum variable will assume possible values within the constants of the enumeration set. This means that the allowed values of the variable is decided by the solver according to the current constraints, yet will contain some (or all) of the constants of the enumeration set.

The following code snippet shows the definition of some primitive type variables:

```plaintext
// Primitives with (default) initial domains
int x0;  // an int variable x0 with initial domain [-inf, +inf]
real x1;  // a real variable x1 with initial domain [-inf, +inf]
bool x2;  // a bool variable x2 with initial domain [true, false]

// Enumerative custom type
enum Speed { "High", "Medium", "Low"};
// a variable x3 with possible values ["High", "Medium", "Low"]
Speed x3;

// Custom type definitions
typedef int [0, 360] Angle;
Angle x4;  // an Angle (int) variable x7 with initial domain [0, 360]
```

Specifically, the first row defines an integer variable `x0` with initial domain \([-\infty, +\infty]\). Follows the definition of a real variable `x1` with initial domain \([-\infty, +\infty]\).
and a boolean variable \(x_2\) with initial domain \(\{true, false\}\). An enum type, called Speed, is defined for allowing the creation variables representing the speed as, for example, the \(x_3\) variable, whose initial domain is the set \(\{High, Medium, Low\}\). Finally, a typedef called Angle is defined as an integer whose initial domain is within bounds \([0, 360]\). The code snippet is closed with the definition of an \(x_4\) variable of type Angle. Since none of these variables is subject to any constraint, their domain, at the end of the execution of the code snippet, will remain untouched.

### 6.1.3 Operators

Once introduced to variables and constants, we can begin to operate with them by using operators. We use operators to impose constraints on declared variables. The complete list of operators is described in the following.

**Assignment operator (\(=\))** The assignment operator assigns a value to a variable. For example

\[
x = 5; \\
y = [0, 20];
\]

assigns the value 5 to the variable \(x\) and the domain \([0, 20]\) to the variable \(y\). The assignment operation always takes place from right to left, and never the other way around. For example

\[
x = y;
\]

assigns the value \(y\) to variable \(x\). The value of \(x\), at the moment this statement is executed, is lost and replaced by \(y\).

It is worth noticing that we are assigning \(y\) to \(x\) therefore, if \(y\) changes at a later moment, it will reflect on the value taken by \(x\) and the other way around. Variables \(x\) and \(y\) will represent exactly the same object after this assignment statement is executed.

The assignment operator can be used, also, during variable declaration for assigning to variables an initial domain through the syntax \(<\text{type}> <\text{id}> = <\text{expr}>\).

**Arithmetic operators (\(+, -, *, /\))** Operations of addition, subtraction, multiplication and division correspond literally to their respective mathematical operators. The semantic, however, is taken by interval arithmetic. Specifically, arithmetic operations are defined as:

- **Addition:** \([x_0,x_1] + [y_0,y_1] = [x_0 + y_0, x_1 + y_1]\)
- **Subtraction:** \([x_0,x_1] - [y_0,y_1] = [x_0 - y_0, x_1 - y_1]\)
- **Multiplication:** \([x_0,x_1] \times [y_0,y_1] = [\min(x_0y_0, x_0y_1, x_1y_0, x_1y_1), \max(x_0y_0, x_0y_1, x_1y_0, x_1y_1)]\)
- **Division:** \([x_0,x_1] / [y_0,y_1] = [\min(x_0/y_0, x_0/y_1, x_1/y_0, x_1/y_1), \max(x_0/y_0, x_0/y_1, x_1/y_0, x_1/y_1)]\)

For example:
CHAPTER 6. THE RIDDLE LANGUAGE

\[ x = 5 + y; \]

assigns to the variable \( x \) the expression \( 5 + y \). Suppose the domain of variable \( y \) is \([10, 20]\), the domain of variable \( x \) will be \([15, 25]\) after the execution of the statement.

It is worth noticing that, similar to what happens for the simple assignment case, we are assigning the expression \( 5 + y \) to \( x \) therefore, if \( y \) changes at a later moment, it will reflect on the value taken by \( x \) and the other way around. Specifically, the variable \( x \) and the expression \( 5 + y \) will represent exactly the same object after this assignment statement is executed. As a consequence, if the domain of \( y \) becomes, for example, \([15, 20]\), the domain of \( x \) will become \([20, 25]\). The value of \( x \), at the moment this statement is executed, is lost and replaced by the expression \( 5 + y \).

Relational and comparison operators (\( ==, !=, >, <, >=, <= \)) Two expressions can be compared using relational and equality operators to know, for example, if two values are equal or if one is greater than the other. The result of such an operation is a boolean variable representing the validity of the relation.

Be careful! The assignment operator (operator \( = \), with one equal sign) is not the same as the equality comparison operator (operator \( == \), with two equal signs); the first one \( (=) \) assigns the expression on the right-hand to the variable on its left, while the other \( (==) \) compares whether the values on both sides of the operator are equal. Consider the following code snippet:

```plaintext
x = 5;  // assigns 5 to x
x = 7;  // assigns 7 to x
x == 7; // compares x with 7 returning true
x == 5; // compares x with 5 returning false
x == [6, 8]; // compares x with [6, 8] returning {true, false}
```

The first statement assigns value 5 to variable \( x \). The second statement assigns value 7 to variable \( x \). The third statement compares the variable \( x \) with the value 7 returning a boolean constant \( true \). The fourth statement compares the variable \( x \) with the value 5 returning a boolean constant \( false \). Finally, the fifth statement compares the variable \( x \) with the domain \([6, 8]\) returning a boolean variable with domain \( \{true, false\} \).

Logical operators (\( !, &, |, ^\) ) Logical operators return boolean variables representing the validity of the operator. To begin, the operator \( ! \) is the domain description language operator for the Boolean operation NOT. It has only one operand, to its right, and inverts it, producing false if its operand is true, and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. The logical operators \( & \), \(| \) and \( ^\) are used when evaluating two (or more) expressions to obtain a single relational result. Specifically, the operator \( & \) corresponds to the Boolean logical operation AND, which yields true if both (all of) its operands are true, and false otherwise. The operator \(| \) corresponds to the Boolean logical operation OR, which yields true if any of its operands is true, thus being false only when both (all of) its operands are false. Finally, the operator \( ^\) corresponds to the Boolean logical operation EXACT-ONE, which yields true if exactly one of its operands is true, thus being false all of its operands are false ore more than one is true.
Assertions  An assertion is a statement that a boolean expression must be true. To assert a boolean expression it is enough to specify the boolean expression as a statement. Notice that enforcing a boolean expression to be true (or, more generally, to false), can result in the updating of the values of the involved variables through constraint propagation. Suppose, for example, we execute the following snippet:

```c
int x = [0, 10];
int y = [10, 20];
bool x_eq_y = x == y;

// Assertion: x_eq_y must be equal to true
x_eq_y;
```

In the above example we are creating an integer variable \( x \) having domain \([0, 10]\), an integer variable \( y \) having domain \([10, 20]\), and a boolean variable \( x\_eq\_y \) having domain \{true, false\}. With the execution of the assertion represented by the fourth statement, however, the value of the variable \( x\_eq\_y \) is constrained to be equal to true. This, in turn, results in forcing the constraint \( x == y \) to be equal to true which, in turn, results in assigning to both variables \( x \) and \( y \) the value 10 (i.e., the only allowed value that makes the two variables equal).

What happens now if we provide an infeasible problem? Suppose, for example, we provide the following code:

```c
int x = [0, 10];
int y = [20, 30];
x == y;
```

The third statement asserts that the constraint \( x == y \) must be true, however the initial domains of the variables \( x \) and \( y \) do not allow the constraint to be satisfied. When these situations occur, we say that we have an inconsistency. The solver detects the inconsistency and returns \(_⊥\). The domains of the variables, after an inconsistency has been detected, are no more valid.

In conclusion, it is worth to notice that the combined use of operators allows to obtain quite complex behaviors. As an example, consider the following code snippet:

```c
// assert linear relations
x0 - x1 > x2 * 3;
x0 != x1;

// assert nonlinear relations
x0 == x2 * x3;

// assert conjunction of relations
x0 + x1 < 2 * x2 & x0 == x2 * x3 & x0 != x4;

// assert disjunction of relations
x0 < 10 | x0 > 100;
```

6.1.4 Complex types

A complex data type (a.k.a. composite data type or compound data type) is any data type which can be constructed using the language’s primitive data types and other complex types. Roughly speaking, a complex type is a group of data elements grouped
together under one name. These data elements, known as members, can have different, either primitive or complex, types.

The best way for defining new types is by means of the `class` keyword. Classes define data types and allow the creation of objects according to characteristics defined inside the class itself. In addition, classes allow fields of any type as well as methods and constructors with any kind of arguments. Classes can be declared in our domain description language using the following syntax:

```plaintext
class type_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    ...
}
```

In order to allow an initialization of the member variables of the type, classes can include a special function called its `constructor`, which is automatically called whenever a new object of the class is created. This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type. Furthermore, when a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor’s body, a colon (:) and a list of initializations for class members. All types have at least one constructor. If a type does not explicitly declare any, the solver automatically provides a no-argument constructor, called the `default constructor`.

The following code, for example, defines a new data type (or class) called `Block` containing an `int` field named `id`.

```plaintext
class Block {
    int id;

    Block(int id) : id(id) {}
}
```

Block b0 = new Block(0);
Block b1 = new Block(1), b2 = new Block(2);

The declared type `Block` is then used for instantiating three objects (variables) called `b0`, `b1` and `b2`. Note how, for creating a new instance of a complex type, the `new` operator is used. Specifically, the `new` operator instantiates a class and, also, invokes the object constructor, returning a reference to the newly created object. Notice that the reference returned by the `new` operator does not have, necessarily, to be assigned to a variable. Indeed, it can also be used directly in an expression.

It is important to clearly differentiate between what is the type name (`Block`), and what is an object of this type (`b0`, `b1` and `b2`). As can be noted by the above example, many objects (such as `b0`, `b1` and `b2`) can be declared from a single type (`Block`).
6.1.5 Type inheritance

Inheritance allows us to define a class in terms of other classes. When creating a class, instead of writing completely new fields and methods, the modeller can designate that the new class should inherit the members of existing classes. Similarly to object oriented programming, we call the existing classes the *base* classes, while the new class is referred to as the *derived* class. The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on. For example, through the code

```java
class HeavyBlock : Block {
    real weight;
    HeavyBlock(int id, real weight) : Block(id), weight(weight) {}
}
```

we create a derived type `HeavyBlock` which *inherits* from the base type `Block`. Since an `HeavyBlock` *is-a* `Block`, all instances of `HeavyBlock` will have a *weight* field of type `real` as well as an *id* field of type `int` which, we say, is inherited from base type `Block`.

Notice that a derived type must explicitly call a constructor of the base class from which inherits. This explicit call, however, can be omitted in the case the base class has a default constructor.

A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class’s base classes (i.e., after the colon). Unless the base classes have a default constructor, the derived class must explicitly call a constructor of each of the base classes.

6.1.6 Existentially scoped variables

Existential quantification is a type of quantifier which can be interpreted as “there exists”, “there is at least one”, or “for some”. Specifically, the domain description language allows the modeler to retrieve a specific instance of a given type which meets certain requirements. Creating an existentially scoped variable can be done, simply, by indicating the type of the possible instances and an identifier for representing the desired instance. For example:

```java
Block b;
```

searches for a block `b` among all the instances of type `Block`. In other words, it creates an object variable, called `b`, whose allowed values are all the instances of type `Block`. Notice that, since `HeavyBlock` is actually a `Block`, the allowed values for the variable `b` will include, also, all the instances of `HeavyBlock`. It is worth to note that, in case no instances exist, the domain of the variable `b` will be empty and the solver will return ⊥ (or, if possible, will backtrack).

The desired requirements are expressed by means of constraints. Consider, for example, the following code:

```java
Block b;
b.id <= 10;
```
In this case the assertion will limit the domain of \( b \) to all the instances of \( \text{Block} \) whose \( \text{id} \) is lower or equal that 10.

It is worth to note that it is possible to use comparison operators on existentially scoped variables. For example

\[
\text{b} \neq \text{b1};
\]

removes the instance represented by \( \text{b1} \) from the allowed values of the variable \( \text{b} \).

### 6.2 Defining predicates

As mentioned in Section 7, the solver reasons in terms of first-order Horn clauses therefore each predicate is associated to a rule that must be complied with in order for the atom, unifying with the head of the rule, to be valid. Consequently, the language has been designed to define both predicates and rules together. Specifically, predicates (and rules) are defined through the following syntax:

\[
\text{predicate} \ <\text{id}> \ (<\text{type}> \ <\text{id}> , \ <\text{type}> \ <\text{id}> , \ldots) \ \{ \\
\text{<rule body>}
\}
\]

in which is represented the identifier of the predicate, a typed list of arguments, and the body of the rule. Suppose, for example, we are interested in representing the location of an agent, we might use the following rule:

\[
\text{predicate} \ \text{At} \ (\text{Location} \ l) \ \{ \\
\}
\]

The arguments of the predicates are considered as existentially quantified variables within the body of the rule which might contain constraints on them and/or on other variables. Specifically, the body of the rule contains a list of statement which is executed for each atomic formula that does not unify. Suppose, for example, we want to express the fact that our agent cannot go on a location outside of the first quadrant of the Cartesian space, we might use the following rule:

\[
\text{predicate} \ \text{At} (\text{Location} \ l) \ \{ \\
1. x >= 0; \\
1. y >= 0;
\}
\]

In some cases, it might be useful to see the body of a rule as the preconditions that must be respected in order for the atomic formula to be valid. The above example can be therefore rephrased as: in order for a robot for being at a given location, the location must be within the first quadrant of the Cartesian space.

Taking advantage of the similarity that a predicate (with its arguments) has a with complex type (with its members) the domain description language gives to the modeler the possibility to define inheritance among predicates, just as is the case of complex types. Just note that the body of the base rules will be executed before the body of the derived rule. The syntax for predicate inheritance is similar to the syntax used for type inheritance.
6.2. DEFINING PREDICATES

Predicates can be also defined within complex types. Each predicate defined within a class has an implicit parameter called \( \tau \), representing the \( \tau \) variable of the corresponding atoms, of the same type within which the predicate has been defined. As an example, the following code defines a predicate \( \text{At} \) with an argument named \( l \) of type \( \text{Location} \) and an implicit argument named \( \tau \) of type \( \text{Robot} \). The associated rule enforces that a robot cannot go on a location outside of the first quadrant of the Cartesian space.

```java
class Robot {
    predicate At(Location l) {
        l.x >= 0;
        l.y >= 0;
    }
}
```

A similar result can be achieved by defining a predicate through

```java
predicate At(Robot tau, Location l) {
    l.x >= 0;
    l.y >= 0;
}
```

The language does not allow the definition of predicates having the same identifier within the same class. We will see soon, in Section 6.2.2, how disjunctions can be explicitly defined within the body of the rules.

### 6.2.1 Representing facts and goals

Once defined predicates and rules, we can now show how to describe facts and goals for our problems. Still following the object-oriented paradigm, we consider facts and goals as objects and, as such, we will create them through the `new` operator. However, since, unlike other languages, we have no means to distinguish facts and goals, we have to distinguish them explicitly through the keywords `fact` and `goal` as a prefix of the identifier. Suppose, for example, we want to express the fact that our agent is at some location, we might use the following statement:

```java
fact at_0 = new At();
```

We can now address the arguments of the fact just as if they are members of the object represented by `at_0`. So we can, for example, put constraints on the coordinates of the locations.

```java
at_0.l.x <= 10;
```

It is worth to note that creating a fact or a goal will, unless specified, create existentially scoped variables for each of its arguments. As a result, the same rule valid for existentially scoped variables, stating that at least one instance of the type must have been previously created, applies also to the creation of facts and goals. Executing the
previous example, without previously creating instances of locations, would result in an inconsistent problem and, therefore, the solver would have returned ⊥.

A convenient method to avoid the creation of existentially scoped variables is to explicitly state the allowed values for an argument. This can be achieved by making use of the syntax `<id>`:<expr>, where `<id>` represents the identifier of an argument and `<expr>` is an expression, within the parenthesis of the new fact or goal. For example

```plaintext
Location 10 = new Location();
fact at_0 = new At(1:10);
```

creates a new location 10 and a new fact at_0 whose parameter l assumes the value 10;

In order to better understand, let us consider a complete example:

```plaintext
// we define a Location class
class Location {
    real x;
    real y;

    Location(real x, real y) : x(x), y(y) {}}

// we define a predicate At
predicate At(Location l) {
    l.x >= 0;
    l.y >= 0;
}

// we create three instances of location
Location 10 = new Location(0, 0);
Location 11 = new Location(1, 1);
Location 12 = new Location(2, 2);

// we create an At fact
fact at_0 = new At();

// we add some constraints on the fact
at_0.l.x <= 1;
at_0.l != 10;

// we create an At goal specifying its l parameter
goal at_1 = new At(1:12);
```

It is worth to note that both facts and goals can be created within the body of a rule. This allows us to define subgoals for the reasoning process. Suppose, for example, we want to express the fact that “All men are mortal”, we might use the following rule:

```plaintext
predicate Mortal (Thing x) {
    goal m = new Man(x:x);
}
```

The creation of scoped formulas (i.e., formulas with a \( \tau \) parameter), either facts or goals, follows a similar syntax. However it is required that the scope is explicitly specified. This can be achieved through the following syntax:

```plaintext
[fact | goal] <id> = new <qualified_id>.(<id>:<expr>, <id>:<expr>, ...);
```
where `<qualified_id>` identifies the scope. The following code, for example, creates two robots and two facts having a different values for their respective `tau` parameters.

```java
Robot r0 = new Robot();
Robot r1 = new Robot();

fact f0 = new r0.At(l:1:10);
fact f1 = new r1.At(l:1:11);
```

Note that the `tau` argument of the fact `f0` is constituted by an object variable whose domain contains the sole robot `r0`. Similarly, the `tau` argument of the fact `f1` is constituted by an object variable whose domain contains the sole robot `r1`.

It is worth to notice that nothing prevents to create facts and goals having existentially scoped variables as `tau` argument. As an example

```java
Robot r0 = new Robot();
Robot r1 = new Robot();

// we create the existential variable
Robot r;

goal g = new r.At(l:1:10);
```

creates a goal `g` whose `tau` has, as allowed values, the two robots `r0` and `r1` (i.e., its allowed values is constituted by the set `{r0, r1}`). This syntax allows the possibility to leave to the solver the responsibility to decide which robot should achieve the `l0` goal.

Finally, in case facts and goals are created within a rule, their scope is inherited from the rule. Suppose, for example, the following rule stating that, in order for a given robot to stay at a given location, the robot must go to that location, the scope of the `gt` goal will be the same of the one that required the application of the rule. Similarly to the above example, the following code leaves to the solver the responsibility to decide which robot should achieve the `l0` goal however, whatever the chosen robot, the same robot will also require to go to the same location.

```java
class Robot {

    predicate At(Location l) {
        goal gt = new GoTo(l:1:1); 
    }
}

Robot r0 = new Robot();
Robot r1 = new Robot();
Robot r;

goal g = new r.At(l:1:10);
```

### 6.2.2 Disjunctions and preferences

Disjunctions constitute a tool offered by the language to express the fact that the solver has to take a choice. As already mentioned above, the language does not allow the
posibility to define predicates having the same name, so, how can we define disjunctions? The language offers the possibility to define disjunctions through the `or` operator. Specifically, the syntax used for expressing disjunctions is the following:

```plaintext
{ ... } or { ... } or ...
```

Whenever the solver encounters a disjunction construct, it non-deterministically creates a branch in the search tree and executes, within each child, the code contained within each block of code (a.k.a. *disjunct*). It is worth noting that disjunctions can appear both in problems and within rules.

Suppose we want a rule stating that, for an agent to be at a given location, it is required either to drive to that location or to fly to the same location. We might express this disjunction within a predicate using, for example, the following code:

```java
define At(Location l) {
    goal d = new DriveTo(l, l);
    } or {
    goal f = new FlyTo(l, l);
}
```

In case it is needed, it is possible to nest multiple disjunctions one inside the other. In other words, nothing prevents to define a disjunction within another disjunction.

Additionally, in order to express preferences, it is possible to assign a cost to disjuncts. Specifically, since the solver receives a penalty for each executed disjunct, it is possible to control the search so as to achieve plans having desired, yet not necessarily possible, characteristics. Costs can be expressed by adding a numeric expression, in square brackets, at the end of the corresponding disjunct. Notice that, in case the cost is not explicitly expressed, a default unitary cost is assigned. Suppose, like in the previous example, we want to state the fact that, for an agent to be at a given location, it is required either to drive to that location or to fly to the same location. However, since driving is cheaper than flying, the former is preferable to the latter. This can be expressed by means of the following code:

```java
define At(Location l) {
    goal d = new DriveTo(l, l);
    } [5] or {
    goal f = new FlyTo(l, l);
    } [200]
```

which give to the solver a penalty of 5, for choosing driving, and a penalty of 200 for choosing flying.
6.3 Representing timelines

The presented basic core is, probably, expressive enough to represent any problem we are interested in (and, given the undecidability of the theory, even some problems in which we are not interested in). It is worth to notice, in this regard, that nothing prevents to use numeric variables as predicate arguments. It might be cumbersome, however, to represent some specific problems for which we have specialized resolution procedures. This introduces, also, some issues related to the efficiency which, probably, would be affected if not using such specialized algorithms.

To this purpose, as already mentioned, it has been introduced a procedure for verifying the consistency of the objects that appear as allowed values for all the \( \tau \) variables of all the atomic formulas. Such consistency check allows the introduction of further constraints (or, more in general, decision points) that will be automatically taken into account by the resolution algorithm. The key point consists in calling different consistency check procedures according to the type of the \( \tau \) variable.

Note that this does not affect the structure of the language in any way. In particular, the language does not require any changes in the case the addition of additional features is demanded. The basic core (and the solver, as well) will remain agnostic of the consistency checking functions, implementing specialized algorithms, provided by possible (future) extension modules. To demonstrate the effectiveness of this approach, however, it will be shown how it applies to the case of the timelines and, specifically, to the most used of them. Specifically, each of the following timeline has its own specific behavior therefore, in order to provide its specialized algorithm, each of these timeline is implemented as a type in a native language (e.g., Java or C++).

First of all, any timeline-based planning problem requires the introduction of two numeric variables which will become always accessible by any code block. Furthermore, two predicates will be introduced for representing temporal impulses and temporal intervals. In all, before the definition of any timeline-based planning problem the following code is sent to the solver:

```plaintext
predicate ImpulsivePredicate(real at) {
    at >= origin;
    at <= horizon;
}
predicate IntervalPredicate(real start, real end, real duration) {
    start >= origin;
    end <= horizon;
    duration == end - start;
    duration >= 0;
}
real origin;
real horizon;
origin >= 0;
horizon >= origin;
```

This will allow us to define some standard behaviors for all the timelines. In particular, this will allow us to easily express temporally scoped assertions. The following sections describe, from a language point of view, how to use some of the timelines.
6.3.1 State-variables

The first timeline that will be presented is the state-variable. As already mentioned in Section 2.3, the semantic of a state-variable is that for each time instant \( t \in T \) the timeline can assume only one value. In order to define a state-variable it is sufficient to define a new derived type whose base type is a StateVariable. All instances of the derived type will be, consequently, state-variables. Similarly, the predicates defined within the new type will be considered as predicates of a state-variable. This allows the modeler to define predicates and rules for the state-variables at domain definition phase.

As an example, the following code creates a type Robot which is a StateVariable.

```java
class Robot : StateVariable {
}
```

Every predicate associated to a state-variable implicitly inherit from the IntervalPredicate, therefore there is no need to define start, end and duration arguments. Furthermore, applying a rule on a derived predicate will result in the application of the IntervalPredicate rule.

As an example, the following code

```java
class Robot : StateVariable {
    predicate At(Location l) {
        duration >= 1;
        goal gt = new GoingTo(1:1, end:start);
    }

    predicate GoingTo(Location l) {
        duration >= 10;
        goal at = new At(end:start);
    }
}
```

defines a Robot, which is a StateVariable, which can navigate between locations. A possible problem can be

```java
Location l0 = new Location(0, 0);
Location l1 = new Location(1, 1);
Location l2 = new Location(2, 2);

Robot r = new Robot();

fact at_0 = new r.At(1:10, start:origin);
at_0.duration >= 1;

goal at_1 = new r.At(1:12);
```

Notice that the goal at_1 cannot unify with the fact at_0 because of the different locations. Calling the consistency check procedure at this stage would put a constraint between the goal at_1 and the fact at_0 (namely, at_1.start >= at_0.end) in order to avoid the overlapping of different values in time, as required by the specific behavior of a state-variable. At this stage, in order to achieve the goal at_1, the solver would execute the body of the rule associated to the At predicate which, in turn, would add the gt subgoal. Finally, the new subgoal would be achieved by executing the code
within the rule associated to the GoingTo predicate which would produce a new sub-
gold at that, at this stage, could unify with the initial fact at_0 leading to a solution
for the whole problem.

Notice that the sequence of the above steps is not related to the language but to
the solver. In particular, choosing when to call the consistency check procedures might
affect strongly the overall performance of the resolution process.

### 6.3.2 Reusable resources

Reusable resources are another predefined type offered by the solver and build on top
of the basic logic core. The semantic of a reusable resource is that concurrent usages of
the same resource cannot exceed the capacity of the resource. Since the sole distinctive
element among the different reusable resources is the capacity, the reusable resource
type is completely defined. Specifically, the reusable resource is defined as a type hav-
ing a single predefined predicate called Use with an amount argument representing the
amount of resource usage. The Use predicate inherits from the IntervalPredicate
the start, end and duration arguments as well as its temporal constraints. Consequently, facts of type Use represent the usage of an amount of a resource in a given
temporal interval. In addition, reusable resources have a single field called capacity
which can be passed to the resource through its constructor. Finally, reusable resources
has a single constructor which instantiates the capacity of the resource.

The application programming interface (API) for a reusable resource is the follow-
ing:

```java
class ReusableResource {
    real capacity;

    ReusableResource(real capacity) : capacity(capacity) {
        capacity >= 0;
    }

    predicate Use(real amount) : IntervalPredicate() {
        amount >= 0;
    }
}
```

Notice that the capacity of the resource is constrained to be greater or equal than
zero. Similarly, the amount of each resource usage is constrained to be between zero
and the resource capacity. Furthermore, as a reinforcement to what has been said till
now, it is worth saying that creating such a type would not be enough for defining a
reusable resource. This is because, in the above code, there is no trace of the specified
algorithm which would avoid the temporal overlapping of too much resource usages.
For this reason reusable resources have been implemented in a native language and not
just in the domain description language.

The following code shows an example of reusable resources usage.

```java
ReusabeResource rr = new ReusableResource(5);

text use = new rr.Use(amount:3, duration:5);
use.start >= 10;
```
6.3.3 Consumable resources

Consumable resources are another predefined type offered by the solver. The semantic of a consumable resource is that, despite productions and consumptions, its level must be within a max and a min value. Associated to consumable resources, two predefined predicates called Produce and Consume each one with its amount argument are intended for representing resource productions and resource consumptions. Similarly to the previous timelines, both the predicates inherit from the IntervalPredicate, therefore there is no need to define start, end and duration arguments nor to introduce temporal consistency constraints. Consumable resources have two fields called min and max, representing the minimum and maximum availability of the resource. Additionally, consumable resources have two fields called initial_amount and final_amount, representing the initial and final amount of the resource. Finally, consumable resources have a single constructor which takes four parameters representing the minimum and maximum availability of the resource and the initial and final amount of the resource.

The API for a consumable resource is the following:

```java
class ConsumableResource {
    real min;
    real max;
    real initial_amount;
    real final_amount;

    ReusableResource(real min, real max,
                     real initial_amount,
                     real final_amount) : min(min),
                        max(max),
                        initial_amount(initial_amount),
                        final_amount(final_amount) {
        min <= max;
    }

    predicate Produce(real amount) : IntervalPredicate() {
        amount >= 0;
    }

    predicate Consume(real amount) : IntervalPredicate() {
        amount >= 0;
    }
}
```

Notice that the minimum availability of the resource is constrained to be lower than the maximum availability of the resource.

The following code shows an example of consumable resources usage.

```java
ConsumableResource cr = new ConsumableResource(-2, 7, 1, 5);

fact p = new cr. Produce(amount:3, duration:5);
p.start >= 10;

fact c = new cr. Consume(amount:1, duration:5);
c.start >= 10;
```
6.3.4 Batteries

Batteries are similar to consumable resources having the minimum availability of the resource constrained to be greater or equal than 0. The behavior, however, is slightly different. Specifically, batteries allow overproductions. In other words it is allowed to exceed the upper limit, however, the surplus is lost. In addition, productions are called charges.

The API for a battery is, therefore, is the following:

```java
class Battery {
    real min;
    real max;
    real initial_amount;
    real final_amount;

    ReusableResource(real min, real max,
                     real initial_amount,
                     real final_amount) :
        min(min),
        max(max),
        initial_amount(initial_amount),
        final_amount(final_amount) {
            min >= 0;
            min <= max;
        }

    predicate Charge(real amount) : IntervalPredicate() {
        amount >= 0;
    }

    predicate Consume(real amount) : IntervalPredicate() {
        amount >= 0;
    }
}
```

6.4 From PDDL to timelines

The following timelines have been introduced to simplify the modeling of classical planning problems. Given the different notion of causality, indeed, it turned out to be useful the introduction of new timeline types which help in the resolution modeled problem.

The overall idea is to introduce a new type, called PropositionalAgent, to represent classical planning agents able to execute both simple and durative actions. The predicates for these agents resemble the actions of classical agents. For example, a classical action like \textit{pick-up}(x - block) is translated into a predicate \textit{PickUp(Block x)}. The body of the rules will contain a goal for each action precondition as well as a fact for each action effect. It is worth to recall that it is possible to put disjunctions within the body of the rules, hence classical planning disjunctions can be easily modeled. Finally, facts and goals are temporally constrained to atoms representing the action in an intuitive way (i.e., preconditions are before the action which is before the
effects and, in case of durative actions, overall conditions are constrained to be during the action).

A second type of timeline, called PropositionalState, is used for representing the state. Also in this case the predicate signature (predicate symbol and its arguments) is easily translatable however, since our framework cannot reason on false facts, we will add a polarity argument to tackle negative atoms. Unlike classical planning, in which atoms that are not not explicitly said to be true are assumed false, the initial state has to be defined extensively by introducing facts and constraining their start argument to be equal to the origin variable. Finally, goals of the problems will be added as goals such that their end argument is constrained to be equal to the horizon variable.

It is worth to notice that constraint among actions and state are not necessarily forced to be as those of classical planning. Specifically, we can easily model situations like states which change some time after an action.

### 6.4.1 Propositional agents

As already mentioned, agent timelines have been introduced for representing classical planning agents able to perform (durative) actions. Specifically, predicates associated to such agents are intended to represent the actions of classical planning agents. Atomic formulas associated to agents can overlap in a free way. Further constraints can be added in order to respect a classical planning rule, called no moving targets, which prevents two actions to overlap when they simultaneously make use of a value and one of the two is accessing the value to update it. In other words, any action which has a precondition $p$ cannot temporally overlap with an action which has an effect $\neg p$. The idea behind this rule is that any reliance on the values at the points of change is unstable. Despite this rule looks anachronistic in a model that takes into account the time, some benchmark domains exploit it to get a desired behavior. Furthermore, since durative actions are kinda like two classical actions separated by a temporal interval, the no moving targets rule applies both to the start and to the end of the actions. Practically speaking, adding the ordering constraints dictated by the no moving target rule to the start and to the end of the actions, if needed, is the main reason for introducing the Agent type.

Similarly to state-variables, in order to define an agent it is sufficient to define a new derived type whose base type is a PropositionalAgent. All instances of the derived type will be, consequently, agents and the predicates defined within the new type will be considered as predicates of an agent. This allows the modeler to define agent predicates at domain definition phase. It is user’s responsability, however, conversely to the state-variable case, to inherit from the IntervalPredicate, in order to represent durative actions, or from the ImpulsivePredicate, for representing classical actions.

The following code shows an example of definition of an agent. The content of the rule is omitted for sake of space.

```java
class BlocksAgent : PropositionalAgent {
    predicate PickUp(Block x) : ImpulsivePredicate {
        ...
    }
}
```
6.4. FROM PDDL TO TIMELINES

6.4.2 Propositional state

The last timeline which has been introduced for representing classical planning problem is the propositional state timeline. This timeline resembles the state of a classical planning problem.

In order to define a propositional state it is sufficient to define a new derived type whose base type is a PropositionalState. All instances of the derived type will be, consequently, propositional states and the predicates defined within the new type will be considered as predicates of an agent. This allows the modeler to define classical planning predicates at domain definition phase. Propositional state predicates implicitly inherit from a new predicate called PropositionalPredicate which has a boolean argument called polarity. This argument is used for representing the polarity of classical planning predicates. Furthermore, PropositionalPredicate implicitly inherit from the IntervalPredicate, therefore there is no need to define start, end and duration arguments.

The PropositionalState type has been provided for adding implicit constraints consisting in the threats defined in Section 2.2. Specifically, ordering constraints will be added between two atoms if their arguments, except for the polarity, unify and the two atoms overlap in time.

An example of propositional state timeline is given by

```plaintext
class BlocksState : PropositionalState {

  predicate HandEmpty() {
  ...]

  predicate Clear(Block x) {
  ...]

  predicate OnTable(Block x) {
  ...]

  predicate On(Block x, Block y) {
  ...]
```
in which the content of the rules is omitted for sake of space.

6.4.3 Putting it all together

Now that we have the basic ingredients to define a classic planning problem, we will see how we can combine them. The basic idea is that we need an agent and a state. In case the :typing requirement is present, the first thing to do is to define types. The definition of types is given directly by the classical problem. We just introduce a basic type called Object, representing the topmost element of the type hierarchy, with an integer member called id. Consider, for example, the 4 Op-blocks world domain in which the only type is block, we can define it as

```cpp
class Object {
    int id;
    Object(int id) : id(id) {}}
class Block : Object {
    Block(int id) : Object(id) {}}
```

The second step is to define the state. Since we need to assign goals to the agent, the state needs a reference to the agent. We will exploit the forward declaration capabilities and will use the agent reference before defining it. So, farther on with the 4 Op-blocks world domain, we have

```cpp
class BlocksState : PropositionalState {
    BlocksAgent agent;
    BlocksState(BlocksAgent agent) : agent(agent) {}}
```

Analogously, since the agent needs to assign goals to the state, it needs a reference to the state. So, continuing with our 4 Op-blocks world domain, we have

```cpp
class BlocksAgent : PropositionalAgent {
    BlocksState propositional_state;
    PropositionalAgent() : propositional_state(new BlocksState(this)) {}}
```
We can now define predicates for our state and for our agent. As already mentioned, the rules for the predicates relative to the state should contain goals on the agent representing the possible actions, properly constrained, for achieving a given goal. On the other hand, the rules for the predicates relative to the agent should contain goals for the preconditions and facts for the effects. We show an example 4 Op-blocks world domain:

```java
class BlocksState : PropositionalState {
    BlocksAgent agent;
    BlocksState(BlocksAgent agent) : agent(agent) {}

    predicate Clear(Block x) {
        duration >= 1;
        [goal put_down = new agent.Put_down(at: start, x:x);
        or [goal stack = new agent.Stack(at: start, x:x);
        or [goal unstack = new agent.Unstack(at: start, y:x);]
        ]
    }
}
```

This, on the other hand, is an example, still in the 4 Op-blocks world domain, of a predicate defined for the agent:

```java
class BlocksAgent : PropositionalAgent {
    BlocksState propositional_state;
    BlocksAgent() : propositional_state(new BlocksState(this)) {}

    predicate Pick_up(Block x) : ImpulsivePredicate {
        goal clear_x = new propositional_state.Clear(polarity:true, x:x);
        clear_x.start <= at - 1;
        clear_x.end >= at;

        goal ontable_x = new propositional_state.Ontable(polarity:true, x:x);
        ontable_x.start <= at - 1;
        ontable_x.end >= at;

        goal handempty = new propositional_state.Handempty(polarity:true);
        handempty.start <= at - 1;
        handempty.end >= at;

        fact not_ontable_x = new propositional_state.Ontable(polarity:false, x:x, start:at);
        fact not_clear_x = new propositional_state.Clear(polarity:false, x:x);
```
CHAPTER 6. THE RIDDLE LANGUAGE

Now that we have defined all the types, the predicates and rules associated to them, we can define the type instances

```
Block a = new Block(1);
Block b = new Block(2);
BlocksAgent agent = new BlocksAgent();
```

as well as the facts and the goals for our planning problem.

```
fact clear_a = new agent. propositional_state. Clear(polarity: true, x:a, start: origin);
clear_a.duration >= 1;

fact clear_b = new agent. propositional_state. Clear(polarity: true, x:b, start: origin);
clear_b.duration >= 1;

fact ontable_a = new agent. propositional_state. OnTable(polarity: true, x:a, start: origin);
ontable_a.duration >= 1;

fact ontable_b = new agent. propositional_state. OnTable(polarity: true, x:b, start: origin);
ontable_b.duration >= 1;

fact handempty = new agent. propositional_state. Handempty(polarity: true, start: origin);
handempty.duration >= 1;

goal on_b_a = new agent. propositional_state. On(polarity: true, x:b, y:a, end: horizon);
```

Notice that the translation from a classical planning problem to a timeline-based planning problem is pretty straightforward, and, as such, can be easily automated.
The reorganization of the timeline-based formalism, already pursued by tLOC and presented in Chapter 3, pushes toward the direction of solving more general problems which might be not strictly related to timelines. The atom concept, for example, being able to not necessarily represent temporal variables, constitutes a more general concept respect to the previously concept of token. Similarly, the type concept represents a superset of the possible timeline types. If on the one hand this increase in generality directly translates into an increase in computational complexity, the solver becomes, indeed, definitely non-decidable, on the other hand, defining heuristics for those problem instances which are decidable becomes a work that can be done, once and for all, for any problem.

The detachment from the strictly related timeline-based problem, for example, could be exploited for the definition of problems for which it does not make sense to reason in terms of temporal evolutions like those related to troubleshooting or to diagnosis systems. Suppose, for example, we have a scooter ignition problem and we want to troubleshoot it. In order to ignite, a scooter needs the gasoline tank to be not empty, the fuel valve to be open and the carburetor to be clean. Additionally, we might know that the probabilities of an empty tank are higher than the probabilities of a closed valve which, in turn, are higher than the probabilities of a dirty carburetor. Finally, we do not initially know whether the tank is empty, the fuel valve is open and the carburetor is clean. We might solve the scooter ignition problem through ORATIO by defining a rule which would explain the system failure as the one represented graphically through a graph in Figure 7.1. Since we do not have information about the state of the system, we introduce three facts with lifted parameters, i.e., EmptyTank(?x), FuelValveClose(?y) and DirtyCarburetor(?z). Finally, we introduce the EngineWontStart() goal and start the ORATIO solving procedure. The graph building procedure would expand the EngineWontStart() goal introducing a disjunction flaw. Since one of the disjuncts contains a EmptyTank(True) subgoal which might unify with the EmptyTank(?x) fact, the graph building procedure terminates, the pruning procedure sets the EmptyTank(False) subgoal at False resulting in a solution in
which the active disjunct is the one containing the $\text{EmptyTank}(\text{True})$ subgoal, suggesting the most probable of the causes as the main issue of the scooter ignition problem. Notice that, in case we knew the gasoline tank is not empty, the initial problem would contain a $\text{EmptyTank}(\text{False})$ fact, the graph building procedure would have reached the $\text{FuelValveClose}(\text{True})$ subgoal suggesting a different, yet still the most probable, cause.

### 7.1 A heuristic for solving constraint logic programming problems

More in general, the proposed critical path heuristic can be exploited, among other things, for solving Constraint Logic Programming (CLP) problems. To the best of the author’s knowledge, indeed, all of the logic programming implementations produce demonstrations (i.e., the analogous of partial plans) in a specific deterministic manner: (a) the next goal to be achieved is chosen according to the order goals arise by following a First-In-First-Out strategy; (b) goals are achieved by applying rules according to the order they are defined. If on the one hand this deterministic behavior can be a great help in debugging programs, on the other it can be an exceptional source of inefficiency. Consider, as an example, the following CLP recursive program for calculating the factorial function.

```prolog
factorial(N,F) :-
    N1 is N-1,
    factorial(N1,F1),
    F is N * F1.
factorial(0,1).
```

Specifically, the above code snippet is aimed at computing the factorial of a number, let us say $n$, as $n$ times the factorial of $n-1$ until the factorial of 0 is asked and 1 is returned. To the best of the author knowledge, there is no CLP implementation which is able to solve the above problem. The main issue with the above implementation, indeed, consists in having defined the base case after the recursive case, resulting in an infinite loop for any query\(^1\). The graph building procedure, as described in Chap-

\(^1\)Seeing is believing: [https://swish.swi-prolog.org](https://swish.swi-prolog.org)
7.2. **FUTURE WORKS**

As already mentioned, since the earliest introduction of heuristics in classical planning, many heuristics have been proposed. It would be interesting to investigate, in addition to the \( h_{add} \) and the \( h_{max} \) case, the applicability and the effectiveness of these more recent heuristics to the timeline-based case. Given their nature, landmark based heuristics, like those described in [67, 89], as well as abstraction based heuristics, like those described in [40] or in [63, 64], represent the ideal candidates for further investigation.

Among the possible future works, additionally, it is worth noting that the proposed approach might be easily applied for solving temporal planning problems. Specifically, rather than distinguishing among \( at_{\text{start}} \), \( at_{\text{end}} \) and \( overall \) conditions, it would be possible to introduce numeric variables into predicates and consider conditions just like preconditions in classical planning problems (enhanced with the proper temporal constraints). Furthermore, as it is typically done in classical planning, it might be interesting in building a similar planner which, rather than searching backward, would apply the rules forward, starting from the body of the rules to the head. Viewing the timeline-based planning problem as a classical planning problem whose operators are the rules allows to exploit hints from classical planning community for solving timeline-based problems. As we have already seen, the main issue that does not allow the use of a classical planner in the resolution of timeline-based problems consists in the presence of numerical variables among the parameters of the predicates and in the presence of numeric constraints within the body of the rules. Specifically, the presence of numerical variables do not allows making predicates ground. A possible workaround might consist in exploiting discretization (in a somehow similar way to [93]). Instead of considering all possible values for a temporal variable, for example, it might be possible to consider discrete times (e.g., a temporal variable might assume three values: \([0, 10]\), \([10, 100]\) and \([100, +\infty)\)). Despite the discretization procedure might be rather expensive, according to the chosen resolution, it would result in a rather straightforward classical planning problem which would be solved in polynomial time. Finally, it might be interesting to investigate other kinds of classical heuristics like those based
on landmarks.

7.3 Conclusions

Planning is a complex task. In the last forty years, a great work in producing a clear formalization and a syntactic standardization of the input/output languages of classical planners has attracted numerous researchers who have produced, especially in the last few years, impressive results. Specifically, the advancements made in the last decade in the techniques for solving classic planning problems, such as domain independent heuristics based on critical-path (e.g., $h_{add}$, $h_{max}$ or $h^{m}$), on delete-relaxation (e.g., $h^{FF}$), on causal graph or on landmarks, have allowed to solve very large instances of problems. The expressiveness of timeline-based planning languages, however, often clashes with the efficiency of problem solving. The poor performances of modern timeline-based planners, indeed, are mostly to be found in the backwardness of the resolutive algorithms which, with a few exceptions, have remained substantially unchanged over the last twenty years.

This thesis offers an initial contribution at reducing the performance gap between classical and timeline-based planners. Despite the gap has not yet been closed and the performance of timeline-based planning still needs to be improved, the main objective of this thesis is to propose an orthogonal view of what causality, in the peculiar declination of timeline-based planning, is. More in general, this thesis wants to propose a starting point for a general approach at the definition of new heuristics for solving reasoning problems defined through more expressive languages which do not rely only on propositional atoms. It turns out that generating such heuristics is itself intractable, yet a cost-effective trade-off can be achieved by exploiting classical planning heuristics and constraint propagation techniques.


This section contains the complete Extended Backus-Naur form (EBNF) of the RIDDLE language.

\[
\begin{align*}
<\text{comp_unit}> & : = ( <\text{type}\_\text{decl}> \\
& \quad | <\text{method}\_\text{decl}> \\
& \quad | <\text{predicate}\_\text{decl}> \\
& \quad | <\text{statement}> )^* \\
<\text{type}\_\text{decl}> & : = <\text{typedef}\_\text{decl}> \\
& \quad | <\text{enum}\_\text{decl}> \\
& \quad | <\text{class}\_\text{decl}> \\
<\text{typedef}\_\text{decl}> & : = '\text{typedef}' <\text{primitive}\_\text{type}> <\text{expr}> <\text{ID}> ';'
<\text{enum}\_\text{decl}> & : = '\text{enum}' <\text{ID}> <\text{enum}\_\text{consts}> \\
& \quad \quad ( ' | <\text{enum}\_\text{consts}> )^* ';'
<\text{enum}\_\text{consts}> & : = '{ ' <\text{StringLit}> ( ',' <\text{StringLit}> )* ' } ' \\
& \quad | <\text{type}> \\
<\text{class}\_\text{decl}> & : = '\text{class}' <\text{ID}> ( ':' <\text{type}\_\text{list}> )^* '{ ' <\text{member}>^* ' } ' \\
<\text{member}> & : = <\text{field}\_\text{decl}> \\
& \quad | <\text{method}\_\text{decl}> \\
& \quad | <\text{constructor}\_\text{decl}> \\
& \quad | <\text{predicate}\_\text{decl}> \\
& \quad | <\text{type}\_\text{decl}> \\
<\text{field}\_\text{decl}> & : = <\text{type}> <\text{variable}\_\text{decl}> \\
& \quad \quad ( ',' <\text{variable}\_\text{decl}> )^* ';'
<\text{variable}\_\text{decl}> & : = <\text{ID}> ( '=' <\text{expr}> )?
<\text{method}\_\text{decl}> & : = '\text{void}' <\text{ID}> ( '(' <\text{typed}\_\text{list}> '? ' )^* '{ ' <\text{block}> ' } ' \\
& \quad | <\text{type}> <\text{ID}> ( '(' <\text{typed}\_\text{list}> '? ' )^* '{ ' <\text{block}> ' } ' \\
<\text{constructor}\_\text{decl}> & : = <\text{ID}> ( '(' <\text{typed}\_\text{list}> '? ' )^* '
\end{align*}
\]
### APPENDIX A. EBNF SPECIFICATION OF RIDDLE

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;init_el&gt;</code></td>
<td><code>:&lt;ID&gt; '(' &lt;expr_list&gt;? ')'</code></td>
</tr>
<tr>
<td><code>&lt;predicate_decl&gt;</code></td>
<td><code>'predicate' &lt;ID&gt; '(' &lt;typed_list&gt;? ')'</code></td>
</tr>
<tr>
<td><code>&lt;statement&gt;</code></td>
<td>`:&lt;assignment_stmt&gt;</td>
</tr>
<tr>
<td><code>&lt;block&gt;</code></td>
<td><code>:&lt;statement&gt;∗</code></td>
</tr>
<tr>
<td><code>&lt;assignment_stmt&gt;</code></td>
<td><code>:(&lt;q_id&gt; '.')? &lt;ID&gt; '=' &lt;expr&gt; ';'</code></td>
</tr>
<tr>
<td><code>&lt;local_var_stmt&gt;</code></td>
<td><code>:&lt;type&gt; &lt;variable_dec&gt; (',' &lt;variable_dec&gt;)* ';'</code></td>
</tr>
<tr>
<td><code>&lt;expression_stmt&gt;</code></td>
<td><code>:&lt;expr&gt; ';'</code></td>
</tr>
<tr>
<td><code>&lt;disjunction_stmt&gt;</code></td>
<td><code>:&lt;conjunction&gt; ('or' &lt;conjunction&gt;)+</code></td>
</tr>
<tr>
<td><code>&lt;conjunction&gt;</code></td>
<td><code>:'[ ' block ' ]' ('[ ' &lt;expr&gt; ' ]')?</code></td>
</tr>
<tr>
<td><code>&lt;formula_stmt&gt;</code></td>
<td>`:'goal'</td>
</tr>
<tr>
<td><code>&lt;return_stmt&gt;</code></td>
<td><code>:'return' &lt;expr&gt; ';'</code></td>
</tr>
<tr>
<td><code>&lt;assignment_list&gt;</code></td>
<td><code>:&lt;assignment&gt; (',' &lt;assignment&gt;)*</code></td>
</tr>
<tr>
<td><code>&lt;assignment&gt;</code></td>
<td><code>:'&lt; '&lt;expr&gt;'</code></td>
</tr>
</tbody>
</table>
| `<expr>` | `:<lit> | '(' <expr> ')' | <expr> ('*' <expr>)+ | <expr> ('/' <expr>)+ | <expr> ('+' <expr>)+ | <expr> ('-' <expr>)+ | '+' <expr> | '-' <expr> | `<q_id>` | (<q_id> '.')? <ID> '(' <expr_list>? ')' | (' <type> ') <expr> | '[' <expr> ',' <expr> ']' | 'new' <type> '(' <expr_list>? ')' | <expr> '==' <expr> | <expr> '>= ' <expr> | <expr> '<=' <expr> | <expr> '>' <expr> | <expr> '<' <expr>`
<expr> ::= <expr> '!=' <expr> |
       <expr> '=>' <expr> |
       <expr> ('|' <expr>)+ |
       <expr> ('&' <expr>)+ |
       <expr> ('^' <expr>)+
<expr_list> ::= <expr> ('.' <expr>)*
<lit> ::= <NumericLit> |
        <StringLit> |
        'true' |
        'false'
<q_id> ::= ('this' | <ID> ('.') <ID>)*
<type> ::= <class_type> |
        <primitive_type>
<class_type> ::= <ID> ('.') <ID>)*
<primitive_type> ::= 'int' |
                   'real' |
                   'bool' |
                   'string'
<type_list> ::= <type> ('.' <type>)*
<typed_list> ::= <type> <ID> ('.' <type> <ID>)*
<ID> ::= ('a'..'z'|'A'..'Z'|'\'')
       ('a'..'z'|'A'..'Z'|'0'..'9'|'\'')*
<NumericLit> ::= [0-9]+ ('.' [0-9]+)? | '.' [0-9]+ |
<StringLit> ::= '"' (ESC1.)* '"' |
ESC ::= '"' | '\\"' | '\\\"'