Temporal and Hierarchical Models for Planning and Acting in Robotics

Arthur Bit-Monnot

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Robotique et Informatique

Présentée et soutenue par :

M. ARTHUR BIT-MONNOT

le vendredi 2 décembre 2016

Titre :

MODELES TEMPORELS ET HIERARCHIQUES POUR LA PLANIFICATION ET L'ACTION EN ROBOTIQUE

Ecole doctorale :

Systèmes (Systèmes)

Unité de recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (L.A.A.S.)

Directeur(s) de Thèse :

M. MALIK GHALLAB

M. FRANCOIS-FELIX INGRAND

Rapporteurs :

M. FEDERICO PECORA, UNIVERSITE D'OREBRO

M. JOACHIM HERTZBERG, OSNABRUCK University Of Applied Sciences

Membre(s) du jury :

M. RACHID ALAMI, LAAS TOULOUSE, Président

M. AMEDEO CESTA, INST SCES COGNITIVES ET TECH ROME, Membre

M. CEDRIC PRALET, ONERA TOULOUSE, Membre

M. DAVID E. SMITH, NASA AMES RESEARCH CENTER MOFFETT FIELD, Membre

M. FRANCOIS-FELIX INGRAND, LAAS TOULOUSE, Membre

M. MALIK GHALLAB, LAAS TOULOUSE, Membre

Abstract

The field of AI planning has seen rapid progress over the last decade and planners are now able to find plans with hundreds of actions in a matter of seconds. Despite those important progresses, robotic systems still tend to have a reactive architecture with very little deliberation on the course of the plan they might follow. In this thesis, we argue that a successful integration with a robotic system requires the planner to have capacities for both temporal and hierarchical reasoning. The former is indeed a universal resource central in many robot activities while the latter is a critical component for the integration of reasoning capabilities at different abstraction levels, typically starting with a high level view of an activity that is iteratively refined down to motion primitives.

As a first step to carry out this vision, we present a model for temporal planning unifying the generative and hierarchical approaches. At the center of the model are temporal action templates complemented with a specification of the initial state as well as the expected evolution of the environment over time. In addition, our model allows for the specification of hierarchical knowledge possibly with a partial coverage. Consequently, our model generalizes the existing generative and hierarchical approaches together with an explicit time representation.

In the subsequent chapter, we introduce a planning procedure suitable for our planning model. In order to support hierarchical features, we extend the existing Partial-Order Causal Link approach used in many constraint-based planners, with the notions of task and decomposition. We implement it in FAPE (Flexible Acting and Planning Environment) together with automated problem analysis techniques used for search guidance. We show FAPE to have performance similar to state of the art temporal planners when used in a generative setting, and the addition of hierarchical information to lead to further performance gain.

Next, we study the usual methods used to reason on temporal uncertainty while planning. We relax the usual assumption of total observability and instead provide techniques to reason on the observations needed to maintain a plan dispatchable. We show how such needed observations can be detected at planning time and incrementally dealt with by considering the appropriate sensing actions.

In a final chapter, we discuss the place of the proposed planning system as a central component for the control of a robotic actor. We demonstrate how the explicit time representation facilitates plan monitoring and action dispatching when dealing with contingent events that require observation. We take advantage of the constraint-based and hierarchical representation to facilitate both plan-repair procedures as well opportunistic plan refinement at acting time.

Acknowledgements

First of all, I would like to thank the École Doctoral Systèmes for financing my PhD and LAAS-CNRS for providing me with an incredibly enjoyable work environment.

I am very grateful to Malik Ghallab and Félix Ingrand, who supervised me during this PhD. They have been great mentors, pushing me forward and allowing me to be independent. I have learned a great deal from theirs skills and expertise through the many discussions we had. Most importantly, their perspective and hindsight taught me to look at the bigger picture and identify the many challenges in AI and Robotics. I would also like to thank Filip Dvorak, for his early work on FAPE. Discovering the field of automated planning, it provided me with a very useful basis to learn about its many specificities.

I would like to sincerely thank the entire RIS team, above all for the friendly and good spirited atmosphere that played a key role in making those three years most enjoyable. On a more technical perspective, my work would not have been possible without all their experience, technical skill and the many tools they developed for interacting with the PR2.

My deepest gratitude goes to all people at NASA Ames Research Center, and especially David Smith, Minh Do and Jeremy Frank, for welcoming me during those four months. I learned a lot at their contact, both from our very technical discussions on a white board and from our more general debates on the state our research community. Besides, I am grateful to the École des Docteurs of Toulouse and the École Doctorale Systèmes for their financial help in this endeavor..

A special thanks needs to be made to Gilles, Paul, Séraphin, Romain and all foosball players I have played with. Post-lunch foosball really has been part of the general well-being at LAAS. Beyond the lab borders, my first thanks go to my parents and family for their unconditional moral support and for giving the will and means to undertake this PhD. A million thanks to Aude and Johann for their constant presence and support in all unforeseen developments as well as for our many gastronomic reunions. Many thanks to Paul, Marie, Gilles, Anne-Isa, Émeline, Lambert and all participants in our regular game and beer meetings.

Last but not least, my foremost thanks go to my spouse Aziliz. Starting our PhD at the same time, we anticipated many difficulties. Instead, those three years turned out to be filled with joy and satisfaction thanks to the great support, understanding and care of Aziliz. The most joyful moment of this PhD being of course the birth of our amazing son Solal.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

Since, the early days of AI, planning has been a perfect example of the strength of model based reasoning in dealing with complex problems. Indeed, the planning community has developed a large number of automated planners that, given a formal description of planning problems, are able to solve complex tasks that were largely beyond the reach of automated systems a few decades ago. However, the field of automated planning is also a vibrant example of the pitfalls of model based reasoning where a tool is only useful when the task at hand meets the assumption of the representation. While automated planners have been extremely efficient at finding plans in very large state spaces, this development has been at the cost of many restrictive assumptions. More importantly, the focus of the field has been for a very large part on the creation of heuristics for reasoning on causal relationships whose efficiency conditions the usefulness of planners.

The classical planning framework has seen many extensions, especially regarding time, which is of particular interest for this dissertation. Most of the work in domain-independent temporal planning has however been focused on an incremental extension to classical planning and often suffers from the same limitations. While some planning models, such as ANML, were proposed to encode much richer temporal problems, they have only seen limited adoption from the planning community.

While being able to reason efficiently on causal relationships is indeed a key ability for generating sound strategies to achieve a desired goal, one must not forget that it is only part of the much more general one of actuating an autonomous agent in the real world. Robotic actors offer a good illustration of the problematics of actuating the increasingly complex robotic platforms. Indeed, while having high level deliberation capabilities such as task planning are highly desirable for autonomous agents, such capabilities are hardly limited to a simple input-output function in a universal model. For acting in the real world, a robotic actor should be able to process the data coming from its sensors and reason on the results of its observations. He must furthermore be able to reason on the limitation of its perception capabilities and on the possibilities to enhance them through specific sensing actions. He should model the contingent nature of the environment and of its own actions to provide a robust strategy that accounts for them. He should detect discrepancies between the actual evolution of the world and the one entailed by its model and react to them by adapting its behavior. He must reason on its spatial environment including its own kinematic limitations. He must transform abstract actions into a series of motor commands that achieve the desired purpose. He must obey to various domain-specific constraints such as social-norms or safety rules. Finally, he must be able to formulate long term strategies where precise knowledge regarding the actual environment is yet unknown.

Fortunately, some of those capabilities can be integrated in a hierarchical fashion where each layer provides a more high-level view of the problem at hand, e.g., by going from an image stream to a characterization of some physical object through given properties until the identification of particular object instances in the agent's model. For such capabilities, it is in general sound for a task planner to keep a high-level view of

the domain. For many other capabilities, producing useful plans requires taking into account domain-specific constraints that cannot be naturally encoded in the usual model of automated planners.

Let us consider the simple example where we have guests arriving for dinner and a service robot that should clean up the apartment, cook dinner and greet the guests on their arrival. Planning in such an environment requires handling the complex temporal constraints required for cooking. The ability to reason on the temporal uncertainty that appears in the duration of the robot's own activities (e.g. time needed to make the cake batter) as well as external events (e.g. arrival time of the guests) is also critical in finding controllable plans that are robust to these contingencies. Furthermore, the requirements of complex activities are sometimes easier to represent with procedural knowledge specifying how things should be done than with traditional operators with conditions and effects, e.g., cooking is a perfect example where a large amount of procedural knowledge is available in the form of recipes that would be hardly expressible through traditional planning operators. Even when following procedural knowledge, the system should still be able to reason on the rest of its activities that might be performed concurrently (e.g. to start cleaning up while the batter is resting).

For a planning framework to be successful in this apparently simple domain, it should first be able to reason on causal and temporal relationships of its activities which is supported by many state-of-the-art temporal planners. The need to tackle temporal uncertainty and handle procedural knowledge has however seen limited interest in the planning community. Finally, a planning framework must be adapted for the integration into a more general acting framework that is responsible for filling the gap between the high-level strategies devised by an automated planner and the sensory-motor capabilities of an actuated platform. At the very least, such integration requires that the planning and the acting framework have a common predictive model for the agent's actions that is used both for producing a strategy and monitoring its execution. The planning framework should moreover be able to react to discrepancies by adapting the current plan to an updated strategy.

Outline and Contributions. Our objective in this thesis is to lay the foundation for such a planning and acting framework. We present FAPE (Flexible Acting and Planning Environment) that lays the foundations of such a framework for planning and acting in robotics.

At the core of the system is a predictive model for representing the current state and expected evolution of the environment as well as the capabilities of an autonomous agent (Chapter 2). The model relies on an action model that extends the usual preconditions and effects to a rich temporal setting. Our core contribution in this chapter is the extension of this temporal model to a hierarchical setting. The resulting representation generalizes both generative and hierarchical planning as well as unique combination of those. Action hierarchies provide us with a convenient way of extending a planning problem with domain-specific knowledge regardless of whether it is meant to express procedural knowledge or to enhance the performance of the system.

To reason on this core model, we introduce a constraint-based planning algorithm (Chapter 3). The choice of a constraint based model is motivated by the great versatility that has been demonstrated by such planners to account for domain-specific requirements as well as their natural support for plan-repair. We extend the existing work to provide a general procedure able to plan both in a fully generative setting as well as in a hierarchical one. We introduce several automated analysis techniques to reason on this planning

model both for inferring constraints on the possible plans as well as for deriving domain-independent heuristics. We demonstrate the resulting planner to be competitive with state-of-the-art temporal planners when used in a domain-independent setting; it can benefit from hierarchical knowledge to further improve its performance. A very preliminary version of this work was published in [Dvo+14b; Dvo+14a]. A specific reachability analysis technique, key to the performance of the system, has been the object of dedicated publications [BSD16a; BSD16b]. A global overview of the planner and a characterization of its performance is still unpublished at the time of this writing.

Given this core capability for planning, we demonstrate how it can be extended to account for temporal uncertainty (Chapter 4). More specifically, we develop a procedure that guaranties the controllability of a plan subject to uncontrollable durations and contingent events. We do this while relaxing the usual assumption of total observability of the environment and instead provide techniques to reason on the observations needed to maintain a plan dispatchable. We show how such needed observations can be detected at planning time and incrementally dealt with by considering the appropriate sensing actions. Most of the formalization and techniques in this chapter have been the object of a dedicated publication [BGI16].

Last, we show how this planning framework is integrated in a more general architecture for acting in order to synthesize the high-level behavior of a robotic actor (Chapter 5). This is done be leveraging our rich temporal model in order to control the execution of a plan under the occurrence of contingent events. We further use our action model to monitor the execution of the plan and detect discrepancies between our predictions and the actual evolution of the environment. When such discrepancies are detected, the system is able to recover by locally repairing the plan. The overall system is integrated with specific robotic skills responsible for perception and integration with the platform actuators.

Given the wide range of topics discussed in this dissertation, each chapter has a dedicated section discussing the related work; allowing a more detailed and contextualized discussion to relate and compare our proposed models and techniques to the existing work.

During the first few months of this PhD, a preliminary version of the planning system was implemented together with Filip Dvorak, that mostly consisted in a straightforward adaptation of constraint-based planning techniques to a limited hierarchical setting. While this system provided a useful lookahead over the challenges of combining hierarchical and constraint-based planning, it suffered many limitations, especially regarding a very restricted planning model and a lack of a formal definition of said model that lead to many approximations and inefficiencies both in the planning algorithm and the underlying implementation. For this reason, no parts of the preliminary system remains the current implementation of FAPE and the key capabilities for the efficiency and expressiveness of the system were developed afterwards based on the insights gathered in this preliminary phase. Hence, the author of this dissertation is the main contributor of all aspects developed in this thesis.

The entire system is available under an open source license.[1] It notably includes *(i)* a parser for the supported subset of the ANML language, *(ii)* the entire planning system, *(iii)* dedicated constraint solvers for reasoning on temporal uncertainty under various observability assumptions, and *(iv)* the acting system with further tools to facilitate the communication with ROS based robotic platforms.

[1] Available at `https://github.com/laas/fape`.

Chapter 2

Temporal and Hierarchical Planning Model

Contents

2.1 Introduction

In order to act deliberately, an agent must have a representation of its environment and of the consequences of its action on the real world. Time is a critical aspect of such models as it allows representing the dynamics of the environment and its expected evolution beyond the current situation.

In this chapter, we are interested in developing a model for a planning agent acting in the real world. The environment is described through a set of state variables that depict high-level properties of physical objects, e.g. that a container is on a truck rather that the actual coordinates of the container and the truck. Temporally qualified assertions on those state variables allow to describe both the current situation of the environment and its expected evolution over time. We further provide predictive models for the actions of a deliberate agent that provide their conditions of applicability as well as their expected impact on the environment. We rely on the many developments that have been made in the context of least-commitment and constraint based planners for the definition of rich temporal models and especially on the ANML language [SFC08] that covers most of the above aspects.

Beyond this generative planning model, we provide a way for the domain designer to give a hierarchical view of the capabilities of an acting agent through a notion of tasks and associated recipes that specify how a given task should be fulfilled. This (optional) construct allows to further extend the planning domain with additional knowledge in order to enforce domain-specific constraints or provide procedural knowledge. For the purpose of generalizing both hierarchical and generative planning, we introduce a notion of *task dependency* that relates a given action to higher level actions in the task network. The resulting representation allows the encoding of both generative and hierarchical problems as well as a unique combination of the two.

The relationship of this model with related work in the field of automated planning in discussed in the last section of this chapter to allow for a better comparison (Section 2.6). We especially discuss the key differences between the proposed model and the one of ANML regarding their hierarchical constructs (Section 2.6.3).

2.2 Main Components

Example 2.1 (Running Example). For the purpose of illustrating this chapter we introduce an example planning domain: a restricted version of the one of controlling a fleet of automated trucks and cranes to transport containers in a harbor introduced by Ghallab, Nau, and Traverso [GNT04].

We represent a harbor with several locations corresponding to specific *docks*. The harbor is served by several automated *trucks* and *cranes* that move containers between different *docks*. We are mainly interested in three primitive actions in this example:

- First, a truck r can move from a dock d_1 to another dock d_2 if they are `connected`. We further consider that an area can contain a single truck and as a consequence, a truck cannot enter an area that is currently occupied by another truck. The duration of this action depends on the distance between the two docks.

- A container c can be loaded onto a truck r if they are both in the same dock d in which a crane is available to perform the task.
- Likewise, a container c can be unloaded from a truck r to a dock d if it is currently on the truck and the truck is in the dock d where a crane is available to perform the task.

We are interested in planning the load and unload operations of *cargo ships* that will be ***docked*** at specific location for a limited amount of time.

2.2.1 Time Representation

We use a quantitative time representation based on time points. We rely on temporal variables (e.g. t, t_1), each designating a time point. Temporal variables are constrained through the usual arithmetic operators that can be used to specify absolute (e.g. $t \geq 9$) or relative constraints (e.g. $t_1 + 1 \leq t_2 \leq t_3 - 2$). Temporal variables are to be attached to specific events of the plan such as the start of an action or the instant at which a given condition must be fulfilled. We rely on a discrete time model, with temporal variables taking a value in the set of positive integers.

2.2.2 Variables and Constraints

We consider a finite set of domain constants O. An example of such domain constants are specific docks or trucks in Example 2.1. We define a *type* as a subset of O, whose elements share a common property. For instance in Example 2.1, we would have a type $\texttt{Docks} = \{\, \texttt{dock1}, \texttt{dock2}, \texttt{dock3} \,\}$. A type can be composed from other types by union or intersection (e.g. $\texttt{Vehicles} = \texttt{Cargos} \cup \texttt{Trucks}$).

An *object variable* x with type T is a variable whose domain $dom(o)$ is a subset of T. A *numeric variable* i is a variable whose domain is a finite subset of the integers.

A *constraint* c over a set of variables $\{\, x_1, \ldots, x_n \,\}$ is a pair $(\overline{x}, \gamma)$ where:

- $\overline{x} = \langle x_1, \ldots, x_n \rangle$ is the sequence of variables affected by the constraint.
- $\gamma \subseteq dom(x_1) \times \cdots \times dom(x_n)$ is the relation of the constraint, giving the allowed sequences of values for this sequence of variables. A table representation of such a relation $\gamma_{\textit{travel-time}}$ is given in Table 2.1 for a constraint representing the travel time between two locations.

To allow for easier representation, we denote the constraint $(\langle x_1, \ldots, x_n \rangle, \gamma_R)$ as $R(x_1, \ldots, x_{n-1}) = x_n$. For instance, the constraint $\textit{travel-time}(\texttt{dock2}, d) = \delta$ is satisfied according to the relation $\gamma_{\textit{travel-time}}$ (Table 2.1) if the object variable d takes the value `dock3` and that the numeric variable δ takes the value 9.

For simplicity, we assume the relation underlying each constraint to be given explicitly, meaning that all allowed tuples are known a priori. In general though, the relation might be implicit, e.g., for a difference constraint.

Numeric variables can also appear in temporal constraints. For instance, $\textit{travel-time}(d_1, d_2) = \delta \wedge t_s + \delta \leq t_e$ enforces a delay δ between the timepoints t_s and t_e whose value is constrained by the time needed to travel from a location d_1 to a location d_2.

Origin	Destination	Travel Time
dock1	dock2	7
dock2	dock3	9
dock3	dock1	6
dock3	dock2	8
	...	

Table 2.1: Table representation of the relation $\gamma_{travel\text{-}time}$ of a constraint representing the travel time between two locations ($travel\text{-}time : \texttt{Docks} \times \texttt{Docks} \rightarrow \mathcal{N}$).

2.2.3 State variables

The evolution of the state over time is represented by multi-valued *state variables* similar to those of SAS+ planning [BN95]. A state variable maps time and a set of objects to an object. For instance $\texttt{loc} : \texttt{Time} \times \texttt{Robots} \rightarrow \texttt{Docks}$ gives the position of a robot over time. The time parameter of state variables is usually kept implicit and we say that the state variable $sv(x_1, \ldots, x_n)$ has the value v at time t if $sv(t, x_1, \ldots, x_n) = v$. A complete definition of the state of the environment at a given time is specified by taking a snapshot of all state variables at this moment.

A fluent $f = \langle sv{=}v \rangle$ represents both a state variable sv and its value v. The fluent f is said to hold at time t if sv has the value v at time t.

2.2.4 Temporally Qualified Assertions

We use temporally qualified assertions to express knowledge or constraints on the evolution of a state variable.

A (temporally qualified) *persistence* assertion, noted $\langle [t_s, t_e]\, sv{=}v \rangle$, requires a state variable sv to keep the same value v over a temporal interval $[t_s, t_e]$. When considered in planning models, persistence assertions are typically used to express goals or conditions for the applicability of an action. For instance, the persistence assertion $\langle [400, 500]\, loc(\texttt{r1}){=}\texttt{dock2} \rangle$ can represent the objective that the robot `r1` is at the dock `dock2` at time 400 and stays there until time 500. We also allow a persistence assertion to be defined at an instant t (noted $[t]$) rather than an interval.

A (temporally qualified) *change* assertion, noted $\langle [t_s, t_e]\, sv : v_1 \mapsto v_2 \rangle$ asserts that the state variable sv will change from having the value v_1 at time t_s to having value v_2 at time t_e. It expresses knowledge on the evolution of the environment, whether it results from the proper dynamics of the environment or as a consequence of an agent's activity. When considered in a planning model, a change assertion typically represents a condition that should be met (sv should have the value v_1 at time t_s) and an effect of some process (sv will have the value v_2 at time t_e). Over the temporal interval $]t_s, t_e[$ the value of sv is undefined. This last property is necessary to represent durative change on state variable with discrete domains without explicitly defining transitional values. For instance, the change assertion $\langle [100, 150]\, loc(\texttt{r1}) : \texttt{dock1} \mapsto \texttt{dock2} \rangle$, means that the robot will move from `dock1` to `dock2` over the temporal interval $[100, 150]$. Its location is undefined over $[101, 149]$: it might be in any dock (including `dock1` and `dock2`) or in a location that has not been explicitly modeled such as the road network.[1]

[1] Additional knowledge regarding the value of a state variable can typically be found by sensing at acting time. Furthermore, a more specialized system can lift some of the lack of knowledge on this

A (temporally qualified) *assignment* assertion, noted $\langle [t]\, sv := v \rangle$, asserts that the state variable sv will take the value v at time t. For instance the assertion $\langle [0]\ \texttt{loc(r1)} := \texttt{dock1} \rangle$ states that the robot `r1` is in location `dock1` at the initial time. An assignment $\langle [t]\, sv := v \rangle$ is a special case of change assertion $\langle [t-1, t]\, sv : any \mapsto v \rangle$ where *any* is an unconstrained variable that can take any value.

While all the previous examples involved only domain constants and absolute times, assertions can of course involve object or temporal variables. The objective of a planning process is then to find activities to be performed and sufficient constraints on these variables such that the expected evolution of the system is both plausible and desirable.

Given a temporally qualified assertion $\alpha = \langle [t_s, t_e]\, sv = v \rangle$ or $\alpha = \langle [t_s, t_e]\, sv : v_1 \mapsto v_2 \rangle$, we denote t_s and t_e as $start(\alpha)$ and $end(\alpha)$ respectively.

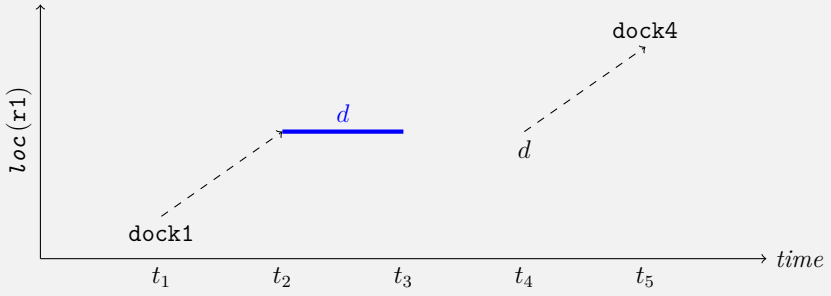
2.2.5 Timeline

A timeline is a pair $(\mathcal{F}, \mathcal{C})$ where $\mathcal{F}$ is a set of temporal assertions on a state variable and $\mathcal{C}$ is a set of constraints on the object and temporal variables appearing in $\mathcal{F}$. A timeline gives a partial view of the evolution of a state variable over time. For instance, the following timeline describes the whereabouts of a robot at different points in time.

Example 2.2. Timeline containing three temporally qualified assertions on the state variable $\texttt{loc(r1)}$.

$$\langle \{\, [t_1, t_2]\ \texttt{loc(r1)} : \texttt{dock1} \mapsto d,\ [t_2, t_3]\ \texttt{loc(r1)} = d,\ [t_4, t_5]\ \texttt{loc(r1)} : d \mapsto \texttt{dock4} \,\}$$
$$\{\, t_1 < t_2 < t_3 < t_4 < t_5,\ \mathit{connected}(\texttt{dock1}, d),\ \mathit{connected}(d, \texttt{dock4}) \,\} \rangle$$

The robot `r1` is at `dock1` at time t_1. Over the time span $[t_1, t_2]$, it will move to a yet unspecified dock d where it will stay until time t_3. Its whereabouts are not constrained over the $]t_3, t_4[$ period. However it must be back in the same dock d at time t_4 from where it will move to `dock4`. Constraints here impose a total ordering on all temporal events and restrict the possible instantiations of d to places connected to both `dock1` and `dock4`.



A timeline typically features temporal and object variables that are not bound yet. As a consequence a timeline can represent different evolutions for the same state variable.

matter. In our example, the uncertainty on the location of the robot will be lifted once its trajectory between the two docks is planned by a specialized path planner. This is however beyond our interest in this chapter and will be discussed in Chapter 5.

On the above example, the state variable will go through different values depending on the value assigned to d. In other cases, some assertions might be allowed to overlap, possibly imposing conflicting values for the state variable. For instance, two persistences $\langle [t_1, t_2]\, sv{=}v \rangle$ and $\langle [t_3, t_4]\, sv{=}v' \rangle$ are conflicting if $[t_1, t_2] \cap [t_3, t_4] \neq \varnothing \wedge v \neq v'$. If those constraints are entailed by $\mathcal{C}$, then the timeline ($\mathcal{F}$,$\mathcal{C}$) containing those assertions would be *inconsistent*; meaning that any valid instantiation of the variables in $\mathcal{F}$ would result in a conflict.

Definition 2.2.1 (Instantiation of a Timeline)**.** A timeline $(\mathcal{F}, \mathcal{C}_1)$ is an instantiation of a timeline $(\mathcal{F}, \mathcal{C}_2)$ if *(i)* all constraints in $\mathcal{C}_2$ appear in $\mathcal{C}_1$ ($\mathcal{C}_2 \subseteq \mathcal{C}_1$), and *(ii)* all variables in $\mathcal{C}_1$ are instantiated.

Definition 2.2.2 (Timeline Consistency)**.** A timeline $(\mathcal{F}, \mathcal{C})$ is (possibly) *consistent* if it has an instantiation $(\mathcal{F}, \mathcal{C}')$ such that $\mathcal{C}'$ is a consistent set of constraints and given $\mathcal{C}'$, the assertions in $\mathcal{F}$ dot not impose, at the same time, conflicting values for the state variable.

Consistency is a desirable property to check since an inconsistent timeline cannot represent a valid evolution of its state variable. An even more desirable property, that we define as *necessary consistency*, is that a timeline cannot contain any conflicting assertion regardless of the choices made when instantiating its variables.

Definition 2.2.3 (Timeline Necessary Consistency)**.** A timeline $(\mathcal{F}, \mathcal{C})$ is *necessarily consistent* if all its instantiations are consistent timelines.

As a consequence, a necessarily consistent timeline is must have a set of constraints such that *(i)* no two change assertions can overlap, *(ii)* no persistence can overlap a change, *(iii)* any two overlapping persistences be on the same value (by definition of a timeline, they are on the same state variable). The timeline of Example 2.2 is necessarily consistent.

Definition 2.2.4 (Causal Support)**.** Given a timeline $(\mathcal{F}, \mathcal{C})$, we say that an assertion $\langle [t, t']\, sv{=}v \rangle$ or $\langle [t, t']\, sv{:}v \mapsto v' \rangle$ in $\mathcal{F}$ is causally supported if there is another assertion $\beta \in \mathcal{F}$ such that either:

- $\beta = \langle [t_1, t_2]\, sv{=}v \rangle$ with $t_1 < t \leq t_2$; meaning that the fluent $\langle sv{=}v \rangle$ is required to hold at an earlier time t_1 and persist until time t; or
- $\beta = \langle [t_1, t_2]\, sv{:}v'' \mapsto v \rangle$ with $t_2 = t$; meaning that the fluent $\langle sv{=}v \rangle$ holds at time t as a result of the effect of an action or of the environment dynamics.

Furthermore all assignment assertions are *a priori supported*.

The persistence assertion of Example 2.2 is causally supported by the first change assertion because *(i)* their temporal intervals meet, and *(ii)* the fluent $\langle \texttt{loc}(\texttt{r1}){=}d \rangle$ achieved is the one required. However, none of the two change assertions are causally supported. This might be surprising regarding the second change assertion because the fluent $\langle \texttt{loc}(\texttt{r1}){=}d \rangle$ is achieved by the first assertion. This fluent is however not constrained to hold during $]t_3, t_4[$ and, as a consequence, its value might still be modified during this interval. This second change assertion could be made causally supported by the addition of a persistence condition ending at t_4 (e.g. $[t_3, t_4]\, sv{=}d$). Another possibility for supporting it would be the introduction a new change assertion $\langle [t, t_4]\, sv{:}d' \mapsto d \rangle$; such an assertion would itself require to be causally supported.

This definition of causal support requires any assertion α to be supported by another assertions whose start timepoint is strictly before the one of α. For all assertions of a timeline to be supported, the earliest one must be an *assignment*, which is by definition *a priori* supported.

2.3 Tasks and Action Templates

Timelines allow to represent a changing environment through the evolution of selected state features over time. A critical part for a planner is a model of how an actor's behavior can influence its environment. For this purpose we now introduce action templates (or actions for short).

Each action has a unique name and a set of parameters. The conditions of applicability as well as the effects of an action are encoded by a set of temporally qualified assertions. A set of constraints restrict the allowed values taken by the action parameters and temporal variables in the action. We augment this usual view of an action with hierarchical features: *(i)* we define a set of tasks symbols $\mathcal{T}$, *(ii)* each action is associated with a task symbol in $\mathcal{T}$ representing the task that this action achieves, and *(iii)* any action can have a set of subtasks representing tasks that must be achieved for this action to provide its desired effects. Furthermore an action can be marked as *task dependent* in which case it cannot be included outside of a preexisting decomposition tree.

Such hierarchical models have been traditionally handled by Hierarchical Task Network (HTN) planners by distinguishing abstract tasks, each associated with one or several methods, and primitive tasks each associated with a single operator (e.g. [EHN94; Nau+03]). A method provides a transformation of an abstract task into a partially ordered set of abstract and primitive subtasks, while an operator represents a primitive from the planner's perspective and provides a predictive model of its effects on the environment. Given an initial task network, HTN planners systematically refine all abstract tasks until only primitive tasks remain and correspond a valid sequence of operators. Unlike most HTN planners, our model does not distinguishes between methods and operators. This is motivated by the remark that hierarchical planners do not treat differently methods from operators: they are are all associated with a task they achieve, all their conditions must be causally supported and all their subtasks (if any) must be refined by another operator or method. From a planning point of view, the distinction lies in an artificial restriction on primitive operators that have no subtasks and are such that no operator achieves the same task as another operator or method. This restriction could usually be lifted without much impact on the algorithm of a hierarchical planner. Methods and operators however carry a semantic difference related to their intended purpose by the domain designer. To facilitate the reading, we will usually refer as an *high-level action*, an action that provide a recipe to achieve a given task, typically expressed as a set of conditions and subtasks. We will refer to as a *primitive action* an action that provides a causal model of its effects on the environment and is intended to be executed through specific commands.

Formally, an action template A is a tuple $(head(A), task(A), dependent(A), subtasks(A), \mathcal{F}_A, \mathcal{C}_A)$ where:

- $head(A)$ is the name and typed parameters of A. The parameter list contains only its object variables and temporal variables are kept implicit. The start and end timepoints of the action are denoted by t_{start} and t_{end} respectively.

- $task(A) \in \mathcal{T}$ is the task it achieves. The general idea is that the presence of A in a solution plan is sufficient to say that $task(A)$ is achieved. This does not depend on whether A is primitive or high level. In the former case, the execution of A will provide the desired effects. In the latter, A will state sufficient conditions for $task(A)$ to be indirectly achieved by other (primitive) actions. A single task can be achieved by multiple actions, each corresponding to a distinct way to fulfill it.

- $dependent(A) \in \{ \top, \bot \}$ is true ($\top$) if the action is *task dependent*. The general intuition (formalized in the next section) is that a *task dependent* action can only be inserted in a plan if it achieves a task whose achievement was required either in the problem definition or through a subtask of another action. If $dependent(A) = \bot$, then A is *free* and can appear in the plan without the need to achieve any task.

- $subtasks(A)$ is a set of temporally qualified subtasks. For any action instance in a plan, a subtask $\langle [t_s, t_e]\, \tau \rangle$ states that the plan should also contain an action instance a that *(i)* starts at t_s, *(ii)* ends at t_e, and *(iii)* has $task(a) = \tau$.

- $(\mathcal{F}_A, \mathcal{C}_A)$ is a consistent set of timelines. $\mathcal{F}_A$ contains conditions for instances of A to be applicable, primarily expressed as persistence assertions. $\mathcal{F}_A$ might also contain change assertions that represent a predicted evolution of the environment as a result of using A. A change assertion $\langle [t_1, t_2]\, \mathtt{sv} : v_1 \mapsto v_2 \rangle$ represents both the condition that $\mathtt{sv}$ has the value v_1 at time t_1 and the effect that $\mathtt{sv}$ will take the value v_2 at time t_2.

Figure 2.1 is an example of the model of a (primitive) action for moving a robot r from dock d to d'. It requires r to be in d at t_{start}, the start time of the action, and d' to be empty at some point t' before the end of the action t_{end}. Its effect is to make the location of the robot be d' at t_{end}, and to have d empty at some point t after t_{start} and before t'. The duration δ of the action corresponds to the travel time from d to d'. The constraints $t_{start} < t_{end}, t_{start} < t$ and $t' < t_{end}$ are implicit in the interval notation. As for primitives in HTN planning, the action has no subtasks and is the only achiever of the eponymous task $\mathtt{move}(r, d, d')$.

$$
\begin{array}{rl}
\mathsf{move}(r, d, d') & \\
\text{task:} & \mathtt{move}(r, d, d') \\
\text{dependent:} & \bot \\
\text{assertions:} & [t_{start}, t_{end}]\ \mathtt{loc}(r) : d \mapsto d' \\
 & [t_{start}, t]\ \mathtt{occupant}(d) : r \mapsto \mathtt{nil} \\
 & [t', t_{end}]\ \mathtt{occupant}(d') : \mathtt{nil} \mapsto r \\
\text{subtasks:} & \varnothing \\
\text{constraints:} & \mathit{connected}(d, d') = \top \\
 & \mathit{travel\text{-}time}(d, d') = \delta \\
 & t_{end} = t_{start} + \delta \\
 & t < t'
\end{array}
$$

Figure 2.1: An action to move a robot r from dock d to dock d'. The action starts at t_{start}, finishes at t_{end} and as duration δ corresponding to the travel time from d to d'. The types of variables are left implicit for readability.

On the other hand, the (high-level) actions of Figure 2.2 give two distinct recipes for achieving the `transport` task of moving a container c to a location d.

m1-transport(c, d)	
task:	`transport`(c, d)
dependent:	$\top$
assertions:	$[t_{start}, t_{end}]\, \texttt{pos}(c) = d$
subtasks:	$\varnothing$
constraints:	$\varnothing$

m2-transport(r, c, d_s, d)	
task:	`transport`(c, d)
dependent:	$\top$
assertions:	$[t_{start}]\, \texttt{loc}(r) = d_s$
	$[t_{start}]\, \texttt{pos}(c) = d_s$
subtasks:	$[t_{start}, t_1]\, \texttt{load}(r, c, d_s)$
	$[t_2, t_3]\, \texttt{move}(r, d_s, d)$
	$[t_4, t_{end}]\, \texttt{unload}(r, c, d)$
constraints:	$\texttt{connected}(d_s, d)$
	$d_s \neq d$
	$t_{start} < t_1 < t_2 < t_3 < t_4 < t_{end}$

Figure 2.2: High-level actions for achieving the task of transporting a container c to a location d. The first one is the phantom action that requires nothing to be done if c is already at its destination d. The second one states that transporting c from d_s to d can be achieved by a sequence of load, move and unload subtasks, using a robot r.

2.4 Chronicles

A *chronicle* is a triple $(\pi, \mathcal{F}, \mathcal{C})$ where π is a partial plan composed of action instances and unrefined tasks while $(\mathcal{F}, \mathcal{C})$ is a set of timelines. A chronicle is a temporal and hierarchical extension of partial plans, as in the plan-space planning approaches.

A planning problem is defined as a tuple $\langle \Sigma, \phi_0 \rangle$ where $\Sigma = (O, \mathcal{SV}, \mathcal{T}, \mathcal{A})$ is the planning domain composed of a set of typed domain objects O, the set of state variables $\mathcal{SV}$, the set of task symbols $\mathcal{T}$ and the set of action templates $\mathcal{A}$. The problem itself is encoded by the chronicle $\phi_0 = (\pi_0, \mathcal{F}_0, \mathcal{C}_0)$ where:

- π_0 is a set of temporally qualified tasks that must be achieved.
- $\mathcal{F}_0$ is a set of temporally qualified assertions that describe the initial state of the environment and its expected evolution together with a set of goals. Some of those are assignment assertions that are *a priori supported*. Assertions that are not a priori supported will typically require the integration of additional actions to have causal support and thus naturally represent goals of the planning problem.
- $\mathcal{C}_0$ is a set of temporal constraints restricting the allowed values for the temporal and object variables in π_0 and $\mathcal{F}_0$. Those can typically be used to express temporaly extended goals, timed initial literals or arbitrary ordering constraints on the tasks to be achieved.

ϕ_0

tasks:	$[t, t']$ `transport(c1, dock3)`
assertions:	$[t_0]$ `loc(r1)`$:=$`dock1`
	$[t_0]$ `loc(r2)`$:=$`dock2`
	$[t_0]$ `on(c1)`$:=$`ship1`
	$[t_0 + 10]$ `docked(ship1)`$:=$`pier1`
	$[t_0 + \delta]$ `docked(ship1)`$:=$`nil`
	$[t_{end}]$ `loc(r1)`$=$`dock1`
	$[t_{end}]$ `loc(r2)`$=$`dock2`
constraints:	$t_0 < t < t' < t_{end}, 20 \leq \delta \leq 30$

The above chronicle represents a planning problem with two robots `r1` and `r2`, initially in `dock1` and `dock2` respectively, and a ship that is expected to be docked at `pier1` at a future interval of time. The problem is to perform a `transport` task of moving a container `c1` initially on `ship1` to `dock3` and to have the two robots in their initial locations at the end. Note that ϕ_0 states a planning problem in terms of tasks, as in HTN, as well as goals, as in generative planning.

The initial chronicle is to be iteratively refined by the planning system until a solution plan is found. We know describe what are those possible transformations and what are the conditions a solution plan must meet.

2.5 Plan: Transformations and Solutions

One difference between HTN planners and generative planners is on the way their plans are generated. HTN planners build plans by systematically decomposing all tasks while generative planers build them by iteratively introducing new action instances. Both processes are interleaved with the insertion of various restrictions of the partial plan, e.g., an ordering constraint between two actions. While this might seem like a simple search procedure, it has important implications on the set of solutions that can be found by each type of planner. Since we aim at combining both approaches, we now define the allowed transformations applicable to an initial chronicle. While this is not meant to restrict the creativity of planning techniques, it is critical to define the set of acceptable solutions of a planning problem.

2.5.1 Task Decomposition

A chronicle $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$ can be refined into a chronicle $\phi' = (\pi_{\phi'}, \mathcal{F}_{\phi'}, \mathcal{C}_{\phi'})$ by decomposing an unrefined task $\tau \in \pi_\phi$ with a new action instance a. This transformation is denoted $\phi \xrightarrow{\tau, a}_D \phi'$ and results in the following ϕ':

$$\pi_{\phi'} \leftarrow \pi_\phi \cup \{ a \} \cup subtasks(a) \setminus \{\tau\}$$
$$\mathcal{F}_{\phi'} \leftarrow \mathcal{F}_\phi \cup assertions(a)$$
$$\mathcal{C}_{\phi'} \leftarrow \mathcal{C}_\phi \cup constraints(a) \cup \{ task(a) = \tau \}$$

This transformation replaces an unrefined task by an action achieving it. The added action can introduce additional unrefined tasks and assertions representing both the conditions and the effects of this action. In addition to the constraints from the action template, we have a constraint $task(a) = \tau$ that ensures that a does refine τ. More

specifically, this constraint unifies all parameters of $task(a)$ and τ and enforces a to start and end at the times specified by the temporal qualification of this task.

2.5.2 Action Insertion

A chronicle $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$ can be refined into a chronicle $\phi' = (\pi_{\phi'}, \mathcal{F}_{\phi'}, \mathcal{C}_{\phi'})$ by the insertion of a *free* (i.e. not task-dependent) action instance a. We denote this transformation by $\phi \xrightarrow{a}_I \phi'$ where ϕ' is composed of the following items:

$$\pi_{\phi'} \leftarrow \pi_\phi \cup \{ a \} \cup subtasks(a)$$
$$\mathcal{F}_{\phi'} \leftarrow \mathcal{F}_\phi \cup assertions(a)$$
$$\mathcal{C}_{\phi'} \leftarrow \mathcal{C}_\phi \cup constraints(a)$$

The important difference with respect to what generative planners do is the restriction to non task-dependent actions. As a result, task dependent actions can only be inserted by decomposing a task and must respect all constraints specified on this task.

2.5.3 Plan Restriction Insertion

A chronicle $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$ can be refined into a chronicle $\phi' = (\pi_{\phi'}, \mathcal{F}_{\phi'}, \mathcal{C}_{\phi'})$ by the insertion of an additional restriction $(\mathcal{F}, \mathcal{C})$ where $\mathcal{F}$ is a set of persistence assertions and $\mathcal{C}$ is a set of constraints over temporal and object variables. We denote this transformation by $\phi \xrightarrow{(\mathcal{F},\mathcal{C})}_R \phi'$ and the resulting chronicle ϕ' is defined as follows:

$$\pi_{\phi'} \leftarrow \pi_\phi$$
$$\mathcal{F}_{\phi'} \leftarrow \mathcal{F}_\phi \cup \mathcal{F}$$
$$\mathcal{C}_{\phi'} \leftarrow \mathcal{C}_\phi \cup \mathcal{C}$$

This transformation does not extend the plan but instead restricts it by either adding persistence conditions or restricting allowed values of some variables. The former is typically used to achieve the causal support of an assertion, similarly to a causal link insertion in plan-space planning. The latter is typically used to remove potential inconsistencies in the partial plan, for instance by imposing an order on two actions with conflicting requirements.

2.5.4 Reachable and Solution Plans

The above transformations define a set of chronicles that can be obtained from the initial problem definition.

Definition 2.5.1. A chronicle ϕ' is *reachable* from a chronicle ϕ if there is a sequence of transformations that turns ϕ into ϕ'.

We note such a sequence of transformations $\rightarrow^*_{D,I,R}$ to represent the fact that they can represent any number of task decompositions, action insertions or restriction insertions. The objective of a planner is usually to find a sequence of transformations such that the resulting chronicle corresponds to a feasible plan.

Definition 2.5.2 (Solution plan)**.** A chronicle $\phi^* = (\pi^*, \mathcal{F}^*, \mathcal{C}^*)$ is a solution to a planning problem (Σ, ϕ_0) if:

- ϕ^* is reachable from ϕ_0,
- π^* has no unrefined tasks,
- all assertions in $\mathcal{F}^*$ are causally supported.
- $(\mathcal{F}^*, \mathcal{C}^*)$ is a set of qnecessarily consistent timelines.

The solution plan is given by the actions in π^* completed with the constraints from $\mathcal{C}^*$. Let us explore some of the consequences of this definition on a solution chronicle ϕ^*. First, because it must be reachable from ϕ_0, the actions in ϕ^* must fulfill all hierarchical constraints. More specifically, if an action is *task-dependent* and was inserted to refine an initial task, then it must respect all constraints placed on it by a higher level action requesting it as a subtask. Second, the fact that all assertions are supported means that all state variables have no unexplained changes. Third, the fact that all timelines are necessarily consistent means that no inconsistency can be introduced in the plan.

It should be noted that this definition does not require all variables to be bound and some choices might be delayed until execution. This is typically the case of temporal variables that can be iteratively instantiated when dispatching.[2]

2.6 Discussion & Related Work

2.6.1 Relationship with other planning paradigms

HTN Planning. Following the definition of Erol, Hendler, and Nau [EHN94], we define an HTN planning problem as a problem whose all solutions can be obtained by a sequence of task decompositions and plan restrictions. Intuitively, this definition requires all actions of the plan to derive from the decomposition of an initial task network and has been the dominating approach for hierarchical planning [Sac75; Tat77; TDK94; WM95; EHN94; Nau+03; Cas+06]. HTN problems can easily be encoded in our model by making all actions *task dependent*, in which case the action insertion transformation is never triggered. Beyond the usual definition of HTN planning that only consider sequential plans [EHN94; Nau+03; GB11; Ber+16; Alf+12], our model fully benefits from the underlying timeline representation that can be used to encode exogenous events, temporally extended goals and more generally all kinds of temporal planning problems including those with required concurrency [Cus+07a] or inter-dependencies [CMR13] (see Example 2.4).

Generative Planning. Similarly, a generative planning problem is a problem where all solutions can be obtained by action insertions and plan restrictions only. Intuitively, this definition rules out any problem in which tasks appear either as the objective or as subtasks of an action in a solution plan. Once again, a purely generative planning problem can easily be expressed by *(i)* making all actions *free*, and *(ii)* forbidding the presence of tasks in any part of the problem. In this setting, the model is similar to the one of IxTeT [GL94] that had a closely related timeline-based representation.

[2]This is mainly possible because the Simple Temporal Networks typically used for this matter describe a convex set of constraint from which a solution can be extracted in polynomial time.

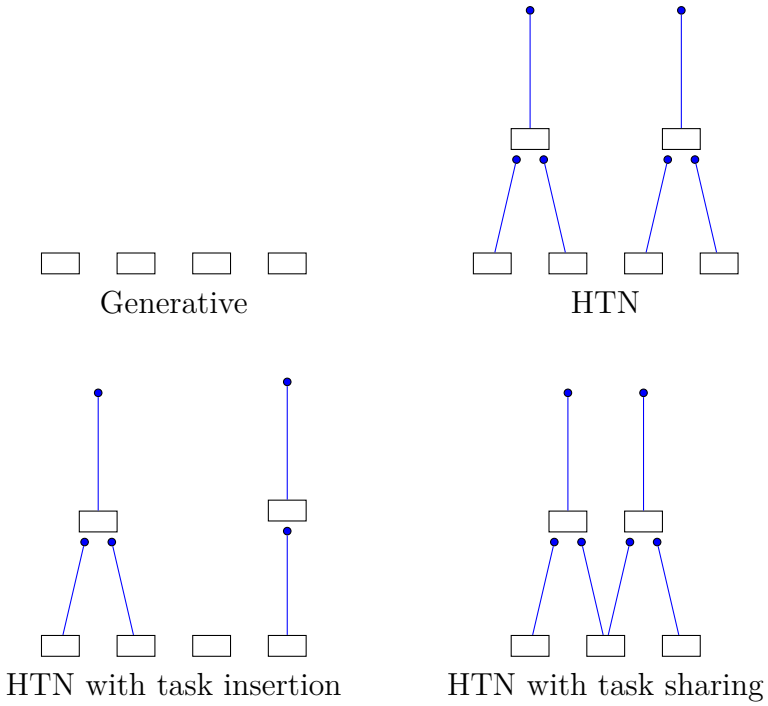


Figure 2.3: Graphical depiction of various planning paradigms. Actions are represented by rectangles and tasks by dots. A task is either part of the problem definition (top level ones) or introduced as a subtask of an action.

No Tasks	**Task dependent actions**		
	None	**Some**	**All**
Generative Planning	HTN planning with Task Insertion	Mixed HTN and Generative Planning	HTN Planning

Table 2.2: Summary of planning problems that can be encoded in our model depending on the presence of tasks and of task dependent actions.

HTN with Task Insertion & Hybrid Planning. Interesting settings to explore are the one at the intersection of HTN and generative planning, that allow both task decomposition and action insertion. Related techniques have been explored under the name of *hybrid planning* [Sch09; CFG00; KMS98; ECW97; MPM94; Bec+14].[3] The common underlying mechanism of those approaches was formalized as *HTN planning with task insertion* that allows the insertion of any operator at any point of the decomposition process [GB11]. The motivating idea was to have the flexibility of generative planners while still benefiting from hierarchical representation to express *(i)* complex constraints between actions, and *(ii)* benefit from improved search performance when guided by methods [ECW97]. However blindly allowing the insertion of any action has negative consequences on both those aspects. First, an important benefit of using HTN planning

[3]The term hybrid planning has more recently been used to denote the use of a plan-space representation by hierarchical planners [BKB14; Bec+14; Bec+15; Ber+16]. We see that as a reaction of the dominance of state-based hierarchical planners (e.g. [Nau+03; Cas+06]) in recent years. However, historically most hierarchical planners relied on a plan-space representation [Wil90; WM95; TDK94]. We do not use this convention as we consider that state-based and plan-based hierarchical planning are to HTN planning what state-space and plan-space planning are to classical planning.

is the ability express constraints between distinct actions by placing them in the same higher level action, e.g., a `rent-a-car` action is always followed by a `return-the-car` action in all task networks it appears in. Hence, being able to insert an action outside of its usual task network reduces the ability of specifying constraints in a hierarchical fashion. Second, action insertion increases the search space of hybrid planners when compared to HTN planners. More specifically, it comes in direct conflict with the planning mechanism of successful forward-chaining hierarchical planners such as SHOP2 [Nau+03]: Indeed in this setting, a planner can decide whether a method is applicable by simple checking that all its preconditions are met by the current state. With task insertion, this simple decision becomes the much more complex one of finding whether there is a sub-plan that would allow achieving the method's preconditions.

Our proposal of *task-dependent* actions allows to tackle those problems by forcing the planners to respect the user-intent that *some* actions should not be used in a generative way. Another way to see it is as the possibility of specifying subparts of the problem that are fully hierarchical such as the one shown in Example 2.3. A related approach to this problem of respecting the user-intent in partially hierarchical problems is tackled by Kambhampati, Mali, and Srivastava [KMS98] by *(i)* associating high-level actions with *primary effects* achieved by their decompositions, and *(ii)* forbidding some state variables to be supported by non-primary effects. To the best of our knowledge this is the only other real *hybrid planner* that distinguishes itself from *HTN with task insertion* by requiring some part of the problem to always obey hierarchical constraints.

Example 2.3. Consider the simple planning problem composed of the *free* move action of Figure 2.1, two *task dependent* actions load and unload and two methods for achieving a `transport` task below.

m1-transport(c, d')

task:	`transport`(c, d')
dependent:	yes
assertions:	$[t_{start}]$ `loc`$(c) = d'$
subtasks:	$\varnothing$
constraints:	$\varnothing$

m2-transport(c, r, d, d')

task:	`transport`(c, d')
dependent:	yes
assertions:	$[t_{start}]$ `loc`$(r) = d$
	$[t_{start}]$ `loc`$(c) = d$
subtasks:	$[t_{start}, t]$ `load`(c, r, d)
	$[t', t_{end}]$ `unload`(c, r, d')
constraints:	$t_{end} - t_{start} < 100$

Figure 2.4: Two high level actions for achieving the task of transporting a container c to a dock d'.

Given a planning problem composed of a single `transport`(`c1`, `dock1`) task any non-empty solution will contain singles load and unload actions and a single truck r will be used. The overall operation is guaranteed to take at most 100 time units. Such constraints are typically hard to represent in generative planning problems and are here possible because all actions but move are task dependent: the m2-transport action can only be inserted to refine the initial `transport` task and its two subtasks are the only way to insert the load and unload actions.

Moreover the hierarchical specification is kept short because the planner is al-

lowed to freely insert a move action to achieve any missing precondition or external requirement on the location of the trucks.

HTN with Task Sharing Another possible characteristic of HTN planners is the ability for task-sharing as formalized by Alford et al. [Alf+16]. In our terminology, it would allow a single action to be a refinement of multiple tasks. This is forbidden in our model since the support of any task requires the introduction of a new action. Stock et al. [Sto+15] use this approach to account for positive interactions between tasks. A very related approach is taken by Georgievski, Lazovik, and Aiello [GLA11] to avoid the introduction of primitive actions already in the plan and whose effects still persist.

Let us consider that we have two tasks τ_1 and τ_2 that both require to transport a container c, initially in d_1, to a location d_2. An HTN planner with task sharing would introduce a first m2-transport(r, c, d_1, d_2) action to support τ_1 and then recognize that it can also be used to achieve τ_2, resulting in a single action being used and no or small branching during search. This simple reasoning has an important flaw since hierarchical planners typically require *phantom* actions that *(i)* are applicable only when the purpose of the task is already completed, *(ii)* do not introduce any additional subtasks (e.g. the m1-transport action of Figure 2.2). These additional actions allow HTN planners to avoid returning plans with unnecessary steps. Moreover, it is also required in HTN with task sharing for completeness; otherwise the planner would always require at least one m2-transport action even if the container was already at its target location As it can be seen in Table 2.3, the presence of phantom actions eliminates the expected gain in the search space as a planner supporting task sharing must now consider one additional case when compared to a classical HTN planner.

Task sharing brings additional complications as it allows interactions between decompositions with a single action being part of multiple decomposition trees. These possible interactions require additional care when writing planning domains, especially when dealing with incomplete models where some conditions for action application are missing. Furthermore, in the case where it is desirable to have a single action refining two tasks, the problem is often best expressed by replacing the task by a condition on a state variable. For instance, allowing the planner to freely insert a move action to achieve conditions on the location of the robot would be a more satisfying encoding than having a phantom move action.

Hierarchical Setting	**Possible refinement for each task**
Idealized HTN with task-sharing	$\tau_1, \tau_2 : \texttt{m2-transport}(r, c, d_1, d_2)$ (refined by the same action)
HTN & phantomization	$\tau_1 : \texttt{m2-transport}(r, c, d_1, d_2) - \tau_2 : \texttt{m1-transport}(c, d_2)$
	$\tau_2 : \texttt{m2-transport}(r, c, d_1, d_2) - \tau_1 : \texttt{m1-transport}(c, d_2)$
HTN with task-sharing and phantomization	$\tau_1, \tau_2 : \texttt{m2-transport}(r, c, d_1, d_2)$ (refine the same instance)
	$\tau_1 : \texttt{m2-transport}(r, c, d_1, d_2) - \tau_2 : \texttt{m1-transport}(c, d_2)$
	$\tau_2 : \texttt{m2-transport}(r, c, d_1, d_2) - \tau_1 : \texttt{m1-transport}(c, d_2)$

Table 2.3: Possible refinements of two $\texttt{transport}(c, d_2)$ tasks denoted as τ_1 and τ_2. Each line maps each task with an (high-level) action refining it. Since m1-transport is a phantom action that does nothing, all refinements lead to an equivalent plan.

2.6.2 Temporal planning models

2.6.2.1 State-transitions vs timelines

STRIPS and PDDL. The original PDDL language [McD+98], along with its ancestors STRIPS and ADL, see actions as state-transition functions all taking a uniform duration. Each action comes with a set of preconditions defining the states in which it can be applied and a set of effects describing the transformation made to the said state. Early work in temporal planning has extended this simple framework by attaching durations to actions and allowing planners to schedule non-interfering actions concurrently. This approach, that we refer to as durative-STRIPS, was notably developed by Smith and Weld [SW99] in TGP and Haslum and Geffner [HG01] in TP4. The definition, for the purpose of the third International Planning Competition (IPC), of PDDL2.1 goes beyond by allowing the placement of effects at an action's start, of conditions at an action's end and of durative conditions that must hold during an action's timespan. While this seems like a minor difference to durative-STRIPS, the resulting language is strictly more expressive as it allows the expression of temporal planning problems with required concurrency [Cus+07a] and with interdependent actions [CMR13] (see Example 2.4).

The philosophy behind PDDL2.1 is to see a durative action as two instantaneous at-start and at-end actions that both produce instantaneous state-transitions. Those "snap" actions are linked together by duration constraints that restrict the possible delays between the start and end times of the action as well as durative conditions that require some condition to hold in all states traversed while the action is executing.

As pointed out by Smith [Smi03], a strong limitation of the language is that conditions and effects can only be placed at the start and end of the action. The authors of the language argue that this limitation can be avoided by splitting a complex temporal action with intermediate time points into multiple subactions [FLH04]. While this process is sound, encoding such durative actions by hand is at best difficult and error prone. In this case, we believe it is useful to have a more expressive temporal language that can be compiled into temporal PDDL to benefit from its large ecosystem. This is the approach taken by Cooper, Maris, and Régnier [CMR10] with their language PDDL-TE or by the authors of ANML [SFC08] on which most of our syntax is based.

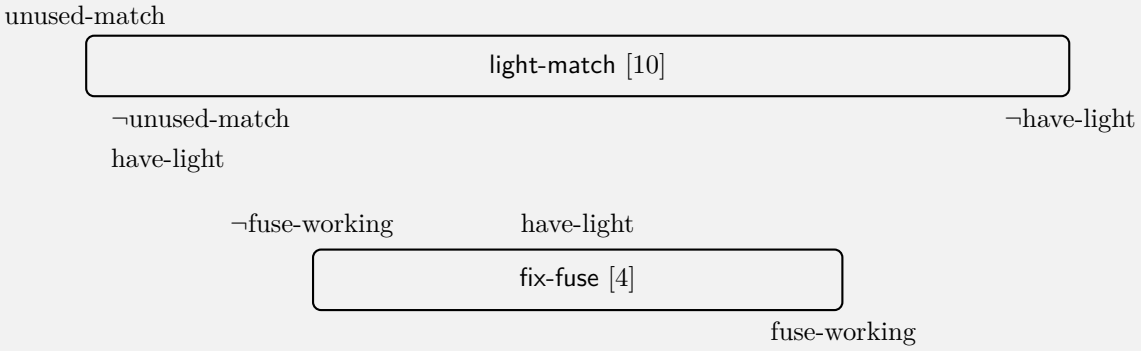
Another important limitation of PDDL2.1 when compared to our model is that a problem definition is composed solely of an initial state and a goal state. Thus, it is not possible to easily express knowledge on the evolution of the environment (e.g. a cargo ship is expected to dock at 11:05) or temporally qualified constraints on the solution plan, (e.g. no truck should be used from 8pm to 6am). Both features were introduced in later versions of the language. Evolution of the environment can be described with the timed initial literals of PDDL2.2 [EH04] that allow truth assignment on predicate at arbitrary times (e.g. at time 1105, the predicate ***docked***(`cargo1`) will become true). Temporally extended goals such as deadlines can be expressed with the state trajectory constraints introduced by PDDL3.0 [GL05]. Even though those extensions are essential in representing real world problems, they have seen a very limited penetration in the temporal planning community: none of the participants of the temporal track in the latest IPC supported them.[4] Instead planners must rely on compilation procedures such as those by Fox, Long, and Halsey [FLH04] to express temporal features of their problems. As a consequence, problems with timed initial literals and temporally extended goals have

[4]See `https://helios.hud.ac.uk/scommv/IPC-14/planners_actual.html` for the supported features of temporal planners.

seen small interest by the planning community. As a testimony, the last two International Planning Competition featured no such problems.[5]

Example 2.4 (Required concurrency & Interdependent actions). We consider the problem of fixing a fuse represented by the two actions below. The light-match action has a start condition requiring the match to be unused (***unused-match***) and immediately produces the effect that the match is not unused (¬***unused-match***) and that the match is burning (***have-light***). After 10 time units (the duration of the action), an end effect is triggered saying that the match is not burning anymore (¬***have-light***).

The fix-fuse action is applicable only when the fuse is initially broken (¬***fuse-working***) and if the match is burning over the entire action (***have-light***). At the end of the action, the fuse is fixed (***fuse-working***).

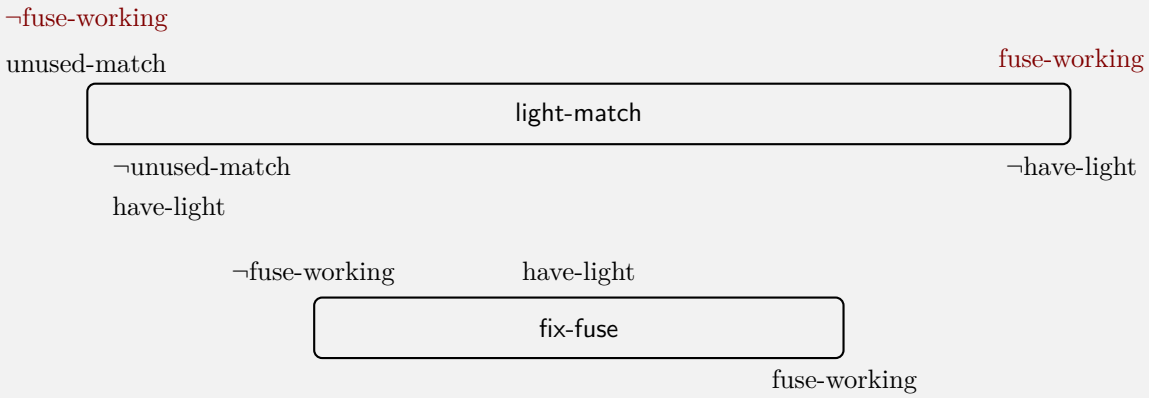


It should be apparent from this model that ***have-light*** will only be true during the execution of light-match. As a consequence, any fix-fuse action must be executed concurrently with the execution of a light-match action.

This scenario where two temporal actions must overlap has been identified as *required concurrency* and is one the main characteristic of temporal planning. Indeed, all PDDL2.1 planning problems with no required concurrency admit a sequential solution plan and the optimal plan (makespan wise) can be obtained by greedily rescheduling a sequential solution [Cus+07a]. Planning models that can feature required concurrency are called *temporally expressive* (e.g. PDDL2.1) and those that do not are called *temporally simple* (e.g. durative-STRIPS).

A more fine grained characteristic of temporal planning languages is the possibility of having interdependent actions as investigated by Cooper, Maris, and Régnier [CMR13]. From our previous example, assume we want to make sure the match is only used in order to fix the fuse. For this purpose, we add the conditions that the light-match action is only applicable when the fuse is broken and that the fuse must be fixed by the end of the action (in red below).

[5]Domains of the 2014 and 2011 IPC can be found at `https://helios.hud.ac.uk/scommv/IPC-14/` and `http://www.plg.inf.uc3m.es/ipc2011-deterministic/` respectively.



The two actions are now interdependent: fix-fuse still requires light-match to produce ***have-light*** and light-match now requires fix-fuse to produce ***fuse-working***. Such problems, and languages that allow them, are denoted by Cooper, Maris, and Régnier [CMR13] as *temporally cyclic*. We usually refer to them as problems with interdependent actions (or interdepencies for short).

Timeline-based representations, such as our own, focuses on scheduling various temporal intervals representing values taken by state variables. In those approaches, actions are typically composed of a set of temporally qualified assertions representing the action's conditions and effects over various state variables (i.e. timelines). Coordination between the different timelines is made by temporal constraints that relate the various assertions of an action.

An early proposal was the one by Allen and Koomen [AK83] based on Allen's temporal algebra [All83]. A timepoint centered view was proposed by Ghallab and Laruelle [GL94] for the IxTeT planner. While many generative and hierarchical planners have since chosen this paradigm [Chi+00b; TDK94; DU11; Ces+09; kR96; FJ03; Bar+12; Mus+02], no dominating language has emerged for the encoding of such planning problems. Given that all models have a very similar notions of causal reasoning, we briefly review the input models of the Europa [Bar+12] and ASPEN [Chi+00b] planning systems because of their historical significance in the creation of ANML that is of more direct interest to us.

Like its predecessor DDL [CO96a; Ces+09], the New Domain Definition Language (NDDL, defined anonymously by Frank and Jónsson [FJ03]) used in Europa has no representation of actions and encodes them as assertions on their own activity timelines. It has no notion of conditions and effects but instead has compatibilities that govern the legal arrangements of values on across timelines (e.g. an assertion representing the activity of moving r to x must be met by an assertion situating r at x). The temporal relations in compatibilities are encoded with a quantitative extension to Allen's Interval Algebra. A solution is a set of timelines that are completely specified (there is no gap between assertions) and that respect all compatibilities.

The Aspen Modeling Language (AML, [Fuk+97; Chi+00b]) is another language based on a timeline representation. The central component of AML are activities that can be used to represent a goal, a high level task or a primitive action. Sub-activities are used to refine goal and high level tasks into primitive actions. This HTN model is completed with a rich syntax for specifying constraints on the state of the system.

Timeline models have been central in the deployment of many Planning & Scheduling

tools for automated mission operation in space, many of which are surveyed by Chien et al. [Chi+12]. As a consequence many such planning system have required the encoding domain constraints that go beyond the usual definition of planning problem, either for performing search control or encoding exotic constraints. This focus on specific problems of many such planners explains to some extent the lack of a unified model.

2.6.2.2 Predicate vs State Variables

As a result of the legacy from STRIPS into PDDL, a lot of planners only support predicate values for encoding state features. We use a state variable representation, similar to the one discussed in [BN95; JB98], to encode state features. State variables have various advantages, one of those being a more natural and concise representation, e.g., it is easier to represent the location of a robot as function mapping one robot to value in a set of n locations than having a n boolean function each saying whether the robot is in a particular location. This conciseness is directly beneficial to state-space planners as they allow for a more compact encoding of the state [DFP10; Hel09].

The state variable representation has the other advantage of making explicit some mutex relations, e.g., a robot can not be in two different locations at the same time. While those are usually not a problem for forward chaining planners, that will never generate a state with two mutex fluents, some plan-space planners, including CPT [Vid11a] and HiPOP [Bec+14; Bec16], do some extra work to identify the mutex relations that are not explicit in their predicate representation.

The large availability of benchmarks in PDDL motivated the need for automated translation from predicates to state variables. Such work was conducted in the context of classical planning by [Hel09] and [DTB13] for use in the Fast-Downward planner [Hel06]. Things are more complex in a temporal setting because one needs to account for interacting concurrent actions to identify mutex fluents. Some of those are studied by Bernardini and Smith [BS11] in the context of PDDL2.2.

Despite those translation techniques, the fact that state variables allow a more natural encoding and result in more efficient planners make a strong case for their presence as first place citizen in our planning model. This is especially true as compilation from state variables to predicates is straightforward and predicates can be represented by state variables with a boolean domain.

2.6.2.3 Continuous vs Discrete Time

The choice of a discrete time representation might seem like a regression given that the most wide spread language for temporal planning support continuous time. Indeed, the specification of PDDL2.1 describes the changes occurring at the start and end of an action as instantaneous. However, Fox and Long [FL03] argue that a platform for executing the plan can only have a bounded precision. For this reason, the PDDL2.1 specification demands that the instant at which an effect is asserted by an action and the instant at which it is required by a condition be separated by a constant delay ϵ. In practical terms, it means that an "instantaneous" effect at time t in fact spans over a temporal interval $[t, t+\epsilon[$ since any condition depending on it cannot occur before time $t+\epsilon$.[6] As noted by

[6]This statement is not true for the original semantics of PDDL2.1 as invariant conditions escaped the bounded precision rule. This inconsistency has been fixed in most practical implementations (e.g. [Col+10; EMR12; RG15; Col+12]). A practical example illustrating this problem is given by Cushing [Cus12, Fig. 3.1].

Cushing [Cus12, Sec. 3.3.3.1], this simple observation is sufficient to make a discretization completeness-preserving by taking as the unit of time the greatest common denominator of all durations.

Thus, our usage of a discrete time representation does not result in any loss of expressivity with respect to the so-called continuous time representation of PDDL. On the other hand, it allows us to avoid the inconsistencies of durative conditions of PDDL. For instance, for a correct support of the bounded precision rule, the duration of a durative condition will depend on the presence of an effect on the same predicate in the action [Col+10; Cus12].

It should be noted that in our model the domain designer can freely choose any unit of time he judges appropriate. Our constraint based representation further allows to let the choice of discretization to the planner by *(i)* having a variable $TimeScale$ whose domain is the set of integers; *(ii)* multiplying all durations by $TimeScale$. The choice by the planner of a value for $TimeScale$ will result in a discretization of time of $\frac{1}{TimeScale}$.

2.6.3 Relationship with ANML

The Action Notation Modeling Language (ANML [SFC08]) is a recent proposal to overcome the absence of apparent causal structure of NDDL and the lack of support for generative planning of AML while retaining some of the simplicity of PDDL. As such, ANML aims at being a unique language for the specification of planning and scheduling problems at NASA.

ANML has a strong emphasize on generative planning with the direct inheritance of earlier timeline based planning models. It comes with a clear notion of action with conditions and effects taking the form of temporally qualified assertions at arbitrary timepoints. While the scope of ANML is larger than the one of interest here (e.g. including resources and conditional effects) the generative part of our model has a direct mapping into ANML.

In addition, ANML provides some facilities for hierarchical planning, mainly as syntactic sugar around traditional conditions and effects. Each action instance is associated with its own predicate that is set to true on the action start and to false on the action end. For example, an instance of an action $A(x)$ is associated with a predicate $A^*(x, id)$ where id is a unique identifier of the instance. The "subtask" assertion $\langle [t, t'] A(y) \rangle$ is simply translated as the condition that there is an i such that $A^*(y, i)$ holds over $[t, t']$; requiring an instance of A with id i to be executing over $[t, t']$. This definition departs from the traditional definition of HTN problems as it allows for task-sharing: a single action can support multiple tasks just like a single effect can support multiple conditions. It also comes with minor inconsistencies for the temporal qualification of tasks: in the previous example, the subaction A is not required to start at t and end at t' but instead should be executing before t and after t'. This difficulty for relating the start and end time of subactions required the introduction of specific temporal qualifiers for the specification of subtasks. Instead we preferred the inclusion of tasks as primary citizens of our planning model. This choice allows us to more clearly relate ourselves to the existing literature on hierarchical planning. Furthermore, we choose not to allow task-sharing. As already mentioned, this facilitates the reasoning on the set of reachable decompositions with no impact on representation capabilities.

The original language definition had no notion of task-dependency and could thus be characterized as an HTN language with task-sharing and task-insertion [Alf+16]. The

latest version of the ANML manual proposes a notion similar to our task-dependency, by the introduction of the keyword *motivated* [Sch+13]. When placed in an action A, an instance a of A can only appear in the plan if there is an action instance b that has a subtask achieved by a and such that $[start(a), end(a)] \subseteq [start(b), end(b)]$. Conceptually, the presence of such an action must be "motivated" by the presence of a higher level action that requires its presence and temporally envelops it. Our *task-dependency* setting differs as it does not require the "motivating" task to be part of an action nor the subaction to be temporally contained by the said action. This simple difference allows us to motivate the presence of actions from tasks placed in the problem definition. This capability is key in relating our model to HTN planning in which all actions are derived from the initial task network.

2.7 Conclusion

In this chapter, we have introduced a planning model that builds on the rich temporal semantics of the ANML language. Namely, the model supports *(i)* temporal actions with assertions placed at arbitrary timepoints, *(ii)* the description of the current state of the environment as well as its expected evolution over time and *(iii)* temporally extended goals.

Beyond those constructs, the model supports an unique mix of generative and hierarchical planning. The key novelty is the notion of task-dependency that allows for a seamless integration of generative and hierarchical models especially allowing for partial hierarchies and the capability of allowing task insertion. The resulting model notably permits the encoding of both generative and HTN planning problems.

To the best of our knowledge, this model is beyond the expressivity of any existing planner. In the next chapter, we will introduce a planning algorithm supporting this representation as well as several dedicated search control methods.

Chapter 3

FAPE Algorithm for Planning with Time and Hierarchies

Contents

3.1 Introduction

In the previous chapter, we have introduced a model to represent temporal planning problems with and without hierarchies. In this chapter, we present a complete planning system that is able to reason on such problems.

Relevant approaches to planning. We start by very briefly sketching the related work in task planning. This overview is meant to present the main concepts that have been influential in the planning community and that we refer to along this chapter. To allow for a more direct comparison and discussion, a detailed review of the related work is delay to Section 3.4.

The field of automating planning has been dominated by forward-search planners, that build a sequence of actions that transform an initial state into a goal state. Such planners search in the space of states, with each action (or sequence of actions) being seen as a state transition function. The key to the efficiency of state space planners is their use of heuristic search to guide their exploration of the set of states. Many heuristics have been devised to evaluate the cost of achieving the goals from a given state by solving a relaxed version of the planning problem. The most influential relaxation is to consider the tractable problem in which all the delete effects of actions are ignored. Planners can compute a heuristic based on this relaxed problem, e.g., FF [Hel06] uses as a heuristic the number of actions in a plan solving this relaxed planning problem. Much work has been done to adapt the state-space approach to temporal planning, e.g., POPF [Col+10] adapts the key ideas of FF to a temporal setting where durative actions are partially ordered and can have many interactions.

The other influential approach in the early days of AI planning has been the plan-space approach. Those planners explore the set of partial plans by iteratively fixing flaws in a partial plan. An example of such a flaw is an open goal: a goal or the condition of an action that has not been achieved yet. Given an open goal, a plan-space planner would extend the plan backward by adding an action which effects achieve the open-goal. A key advantage of plan-space planners is that they naturally yield partially ordered plans that can represent many possible sequences of actions. They also exploit the structure of the problem since they will only consider plans where all actions contribute directly or indirectly to the goals. Despite those advantages, plan-space planners have not benefited from the development of planning heuristic as well as state-space planners. They however allow an intuitive support for temporal planning by simply keeping track of temporal constraints in an STN.

The ideas behind plan-space planning have been further developed into least-commitment constraint-based planners that keep a lifted plan representation in which the parameters of actions are not instantiated. Instead, the planners keep track of the possible values of those parameters as an underlying constraint satisfaction problem. Those planners rely on the plan-space planning mechanism but their lifted representation allows them to keep a more compact search-space where a single search node represents many possible plans. Such planners have been very early adapted to support temporal planning (e.g. IxTeT [GL94]). However, their lifted representation comes as a liability when trying to adapt the usual heuristic from domain-independent state-space planners and they often rely on domain-dependent search control.

In parallel to those generative planners, hierarchical planners have been influential by their capability to benefit from domain specific knowledge encoded in high-level actions. Those planners typically perform a top to bottom decomposition of a set of initial tasks. Like generative planners, they either follow state-based approach by building a sequence of actions from an initial state (e.g. SHOP2 [Nau+03]) or a plan-based approach where actions are partially ordered and require some additional reasoning for handling open goals (e.g. O-Plan [TDK94]). Such planners almost exclusively rely on hand coded rules to guide their exploration of the search space.

Overview of the chapter. In this chapter, we present a planning algorithm based on least-commitment planning. In the next section, we extend the plan-space planning procedure of those planners to account for the presence of hierarchical actions. The procedure is sound and complete for both generative and hierarchical planning problems. In the following section, we introduce various techniques to reason on the search space of the planner. Because of the lifted representation, a key capability is to reason on the possible instantiations of a partial plan which is done through a dedicated reachability analysis technique. We also present heuristics to guide the exploration of the search space both in generative and hierarchical settings. We review and discuss the related work in Section 3.4. Section 3.5 evaluates how the proposed techniques contribute to the overall efficiency of the planning system and draws some comparison with existing temporal planners.

3.2 A Plan-Space Planning Procedure

3.2.1 Overview

Our planning procedure extends the approach of chronicle planning [GNT04, Sec. 14.3] that was first defined in IxTeT [GL94]. The chronicle approach is an extension of plan-space planning to a rich temporal representation. Our main contribution regarding the planning procedure is to extend it to support the association of hierarchical and generative planning.

Given a planning problem with an initial chronicle ϕ_0, the objective of the planner is to refine ϕ_0 into a solution chronicle ϕ^* where all tasks have been refined, all assertions are causally supported and the set of temporal assertions form necessarily consistent timelines. For this purpose, the planner is equipped with the ability to detect *flaws* in a partial plan: features that prevent the partial plan from being a solution. Let us define a *flaw* in a chronicle $(\pi, \mathcal{F}, \mathcal{C})$ as either:

- an *unrefined task* $\tau \in \pi$,
- an *unsupported assertion* $\alpha \in \mathcal{F}$. This flaw is a generalization of the notion of open-goal in plan-space planning.
- a pair of assertions $(\alpha, \beta) \in \mathcal{F} \times \mathcal{F}$ that can be conflicting given $\mathcal{C}$. This flaw is called *conflicting assertions* and again is a generalization of the notion of threats in plan-space planning to a more a general temporal model.

Each type of flaw matches one of the necessary conditions for a chronicle to be a solution plan (Definition 2.5.2). Hence the presence of a flaw in a partial plan means it is not a solution to the planning problem. Intuitively, every flaw must be resolved to transform a partial plan into a solution plan; resolving each flaw will require the application of one or multiple plan transformations (as defined in Section 2.5).

An abstract view of the planning procedure, FapePlan, is given in Algorithm 1. At each step, for a given partial plan, the planner chooses a flaw f to solve. Because all flaws must eventually be solved, this choice is not a backtracking point. Next, the planner faces the choice of resolver ρ to handle the flaw. Each resolver results in a transformation of the partial plan ϕ into a refined partial plan ϕ' in which f is absent.

Our planning procedure follows the general one of refinement planning (as formalized by Kambhampati, Knoblock, and Yang [KKY95] and Schattenberg [Sch09, Sec. 2.6 to 2.8]) where a set of *deficiency detection functions* identify flaws in a partial plan and a set of *modification generation functions* generate modifications of the plan that fix the flaws (i.e. resolvers). Our procedure is an instantiation of this more general scheme with three detection functions (one for each flaw) and their modification generation functions implicitly defined by the set of resolver associated to a flaw. This procedure has been used from early partial order planners (e.g. [PW92; YS03]) to more advanced generative temporal planners [GL94; VG06]. Many hierarchical planners follow a similar approach, including the recent HiPOP [Bec+14] and PANDA [Sch09]. The differences between all those planners lie in four aspects: *(i)* their definition of flaws and resolvers, *(ii)* their internal representation of a partial plan, *(iii)* the type of constraints and propagation used to reason on variables in the plan, and *(iv)* the strategy for exploring the search space. The remaining of this section is dedicated to clarifying the first three aspects. We start by detailing our definition of flaws as they are critical in understanding the planning mechanism at work to mix hierarchical and generative planning. Next, we detail our constraint based representation that follows a least commitment approach where both temporal and object variables are partially instantiated. Temporal constraints are represented as a Simple Temporal Network (STN) while object variables are handled in a dedicated binding constraint network using arc-consistency for propagation. Last, we briefly characterize the search space and study the formal properties of the algorithm, including soundness and completeness.

3.2.2 Flaws and Resolvers

The primary function of flaws is to act as detection functions of termination: a flaw indicates a deficiency of a partial plan that makes it unfit to be a solution. The definition of flaws is critical for the soundness of the planning procedure as the planner consider as a solution plan any partial plan that is flaw-free. Equally important are their associated resolvers that describe how a partial plan can be refined into a solution plan. All transformations applied to a partial plan by the planner are motivated by the resolution of a

Algorithm 1 FapePlan algorithm: returns a solution plan achieving the tasks and goals in ϕ for the domain Σ.

```
function FapePlan(Σ, φ)
    Flaws ← flaws in φ
    if Flaws = ∅ then return φ
    f ← select a flaw in Flaws
    Resolvers ← resolvers for f
    if Resolvers = ∅ then return failure
    nondeterministically choose ρ ∈ Resolvers
    φ' ← Transform(φ, ρ)
    return FapePlan(Σ, φ')
```

flaw and consist of the application of a resolver. Thus, resolvers are as well critical in shaping the search space of the planner. Ill-defined resolvers can easily result in a lack of completeness of the planner when some solution is not accessible through any resolver.

There is another strategic aspect in the definition of resolvers: when solving a flaw, the planner should ideally disregard any resolver that would result in a dead-end, e.g., a resolver that would make all resulting plans inconsistent. This feature does not impact soundness (such inconsistencies would we eventually detected later during search) but allows the planner to better identify dead-ends (i.e. when a flaw has no resolvers) or necessary modification to the plan (i.e. when a flaw has a single resolver).

3.2.2.1 Unsupported assertion

Our first type of flaw is meant to identify the lack of causal support of some assertion in the partial plan. It is a simple extension to the notion of *open goal* in plan-space planning to a more general temporal framework.

Our definition of a solution plan (Definition 2.5.2) requires every assertion to be causally supported. Given a partial plan ϕ, the planner incrementally tracks this property by associating every assertion that requires causal support for a fluent $\langle sv{=}v\rangle$ to another one that produces the desired value. With the exception of a priori supported assertions, the planner considers an assertion $\alpha = \langle[t_1, t_2]\, sv{=}v\rangle \in \mathcal{F}_\phi$ or $\alpha = \langle[t_1, t_2]\, sv{:}v \mapsto v'\rangle \in \mathcal{F}_\phi$ as causally supported iff:

- there is another assertion $\beta \in \mathcal{F}_\phi$ that produces $\langle sv{=}v\rangle$ at an earlier time t_0. The assertion β is said to *produce* $\langle sv{=}v\rangle$ at time t_0 if it is an *a priori* supported assertion $\langle[t, t_0]\, sv{:=}v\rangle$ or if it is a change assertion $\langle[t, t_0]\, sv{:}v'' \mapsto v\rangle$. β is called the *causal supporter* of α.
- the value of sv cannot change during the temporal interval $[t_0, t_1]$, i.e., sv keeps the value v from the end of β until the start of α.

For the planner to incrementally keep track of causal supports, we use a notion of causal link $\beta \to \alpha$ denoting that the assertion β is the causal supporter of α. A causal link is associated with a new persistence assertion $[t_0, t_1]\, sv{=}v$ that fulfills a similar job as the one of causal links in plan-space planning: it prevents any change on the value of the state variable sv from the end of β (t_0) until the start of α (t_1).

Definition 3.2.1. Given a chronicle $(\pi, \mathcal{F}, \mathcal{C})$ we say that an assertion $\alpha \in \mathcal{F}$ is *supported* if either:

- α is *a priori* supported, or
- α has a causal supporter β and an incoming causal link $\beta \rightarrow \alpha$.

With the exception of *a priori* supported assertions, the planner assumes all assertions to be unsupported until a causal link is added to the chronicle. The presence of an unsupported assertion constitutes a flaw that must be solved by enforcing its causal support. The choice of how to support an assertion thus requires finding another assertion β to be its supporter. There are two possibilities: β can either be already in $\mathcal{F}_\phi$ or it can be added through the insertion of another action. Action insertion is more complex than in purely generative planning because we must take into account the hierarchical constraints of the domain. For this reason, our resolvers for an unsupported assertion are more elaborate than those for an open goal in plan-space planning: instead of directly inserting an action that provides a causal support, we instead select or create a decomposition tree in which the supporting assertion is taken.

Possible effects. We say that the fluent $f = \langle sv = v \rangle$ is a *direct effect* of an action a if f is produced by an assertion of a, i.e. a has a assertion of the form $\langle [t, t']\, sv : v' \mapsto v \rangle$ or $\langle [t'']\, sv := v \rangle$.

A fluent f is a *possible effect* of an action a, noted $f \in E^+_a$, if it is either a direct effect of a or a possible effect of the decomposition of one of its subtasks. A fluent is a *possible effect* of a task τ, noted $f \in E^+_\tau$, if f is a possible effect of an action a that refines τ. Possible effects thus represent the different fluents that can be achieved as part of the decomposition tree of a given task or action.

$$E^+_a = \mathit{direct\_effects}(a) \bigcup (\cup_{\tau \in \mathit{subtasks}(a)} \quad E^+_\tau)$$
$$E^+_\tau = \bigcup_{m \,\in\, \{a \,\mid\, \mathit{task}(a) = \tau\}} E^+_m$$

The possible effect f of an action a (resp. a task τ) is also associated with a duration representing the minimal delay from the moment the action (resp. task) starts to the instant at which the possible effect can be produced, noted $\Delta_{PosEff}(a, f)$. This minimal delay is computed by taking an optimistic view of the delays enforced as temporal constraints in the actions, as illustrated in the following example.

Example 3.1. The move(r,d,d') action of Figure 2.1 has the possible effects $\langle \texttt{loc}(r){=}d' \rangle$, $\langle \texttt{occupant}(d){=}\texttt{nil} \rangle$ and $\langle \texttt{occupant}(d'){=}r \rangle$. The first one is produced at time t_{end}, thus its associated delay is the duration of the action $t_{end} - t_{start}$:

$$\Delta_{PosEff}(\mathsf{move}(r, d, d'), \langle \texttt{loc}(r){=}d' \rangle) = t_{end} - t_{start} = \mathit{travel\text{-}time}(d, d')$$

The fluent $\langle loc(r){=}d' \rangle$ is also a possible effect of the action m2-transport (Figure 2.2) because the presence of its move subtask means it can be indirectly produced as a result of inserting m2-transport in the plan. The delay associated to the possible

effect is computed recursively as:

$$\begin{aligned}\Delta_{PosEff}(\mathsf{m2\text{-}transport}(r,c,d,d'), \langle \mathtt{loc}(r){=}d' \rangle) \\ &= \text{min-delay}(t_{start}, t_2) + \Delta_{PosEff}(\mathsf{move}(r,d,d'), \langle \mathtt{loc}(r){=}d' \rangle) \\ &= \text{min-delay}(t_{start}, t_2) + \textit{travel-time}(d,d') \\ &= \text{min-delay}(t_{start}, t_1) + \text{min-delay}(t_1, t_2) + \textit{travel-time}(d,d') \\ &= \textit{duration}(\mathsf{load}(r,d,c)) + 1 + \textit{travel-time}(d,d')\end{aligned}$$

In practice the computation of minimal delays (such as the expansion of min-delay(t_{start}, t_2)) is done by a shortest path computation in an STN containing all timepoints and temporal constraints of the action.

For an unsupported assertion α, there are three different places where a supporting assertion β can be found:

- β can be an assertion already in the plan,
- β can be introduced by decomposing an unrefined task $\tau \in \pi$. This requires that τ has a possible effect supporting α (i.e. $\textit{cond}(\alpha) \in E^+_\tau$).
- β can be introduced through the addition of a *free* action a, where β is either in the assertions of a or introduced by the decomposition of the subtasks of a. This requires a to have a possible effect supporting α (i.e. $\textit{cond}(\alpha) \in E^+_a$).

It is however impractical for the planner to branch on all the different assertions that can be used for causal support. For instance, a single recursive task might produce infinitely many assertions eligible for causal support. Instead, the planner branches on the choice of a source for providing the causal supporter, e.g, for a given unsupported assertion, the planner will commit to select its causal supporter in the set of assertions resulting from the decomposition of a particular unrefined task $\tau \in \pi_\phi$.

Intermediate support constraints. We associate an unsupported assertion α to a set of tasks DT_α representing a commitment on the origin of the supporter for α. More specifically, for any task $\tau \in DT_\alpha$, the assertion supporting α can only be chosen in those introduced by an action appearing in π_ϕ as a descendant of τ. An action is a descendant of τ if it refines τ or refines a subtask of an action descending from τ.

This mechanism allows tying the resolution of an unsupported assertion to a tree or subtree of the task network. DT_α is initially empty and might be extended during search to track commitments made on the origin of the support of α.

If there is an unrefined task $\tau \in DT_\alpha$, the resolution of α is postponed until τ is refined.

Resolvers. Let us now explore the three different resolvers for an unsupported assertion $\alpha = \langle [t_s, t_e]\, sv{=}v \rangle$ (resp. an unsupported change assertion $\alpha = \langle [t_s, t_e]\, sv : v \mapsto v' \rangle$):

- **Direct supporters.** Let DS_α be the set of all assertions $\beta \in \mathcal{F}_\phi$ such that β produces $\langle sv = v \rangle$. For any change assertion $\beta \in DS_\alpha$, a possible resolver for α is to add a causal link from β to α.

The transformation resolving the unsupported assertion α is a simple plan restriction that introduces a causal link from β to α: $\phi \xrightarrow{(\{\beta \rightarrow \alpha\}, C)}_R \phi'$. The additional set of constraints C simply enforces that the value produced by β is the one needed by α. Furthermore, the assertion representing the causal link is *a priori* supported since it is inseparable from β that causally supports it.

For instance, if the assertion $\langle [t_1, t_2]\, loc(\texttt{r1}) : \texttt{d1} \mapsto d \rangle$ was selected to be the causal support of an assertion $\langle [t_3, t_4]\, loc(r) = \texttt{d4} \rangle$, we would have an additional set of constraints $t_2 \leq t_3 \wedge \texttt{r1} = r \wedge d = \texttt{d4}$. The causal link would take the form of the persistence assertion $[t_2, t_3]\ \texttt{loc}(\texttt{r1}) = \texttt{d4}$.

To avoid introducing redundancies in the search space, we must also take into account the previous commitment that have been made regarding the source of the causal support of α. For this reason, we only consider an assertion $\beta \in DS_\alpha$ as a possible resolver if, for any $\tau \in DT_\alpha$, β was introduced by an action descending from τ.

- **Delayed support from existing task.** Another possible source of supporting statements for α are those that will be inserted when decomposing yet unrefined tasks of the partial plan.

  Let TS_α be a subset of the unrefined tasks $\tau \in \pi_\phi$ such that

  - $\langle sv = v \rangle$ is a possible effect of τ (i.e. $\langle sv{=}v \rangle \in E^+_\tau$),
  - there is enough time for the possible effect to occur before α. Put otherwise, it means the addition of the temporal constraint $start(\alpha) - start(\tau) \geq \Delta_{PosEff}(\tau, \langle sv{=}v \rangle)$ does no make the temporal network inconsistent.

  For any unrefined task $\tau \in TS_\alpha$, a possible resolver for α is to add τ to DT_α. While this does not directly resolve the open goal, it makes a commitment to a subset of resolvers: the only allowed supporting assertion for α will be the ones descending from τ. In practice this commitment means that: *(i)* the choice of a supporting assertion for α will be delayed at least until τ is refined; *(ii)* the chosen refinement for τ will need to provide an enabler for α. It should be noted that this resolver does not bring any change to the partial plan itself but constrain the planner a restricted set of solution plans.

  Once again, to avoid redundancies in the search space, we must account for the previous commitments made on the causal supporter of τ. For this reason, we only consider as a resolver a task $\tau \in TS_\alpha$ if it is a descendant of all tasks in DT_α.

- **Delayed support from new free action.** The last possible source of supporting assertions comes from the introduction of free actions that have $\langle sv{=}v \rangle$ as a possible effect.

  If the planner already made a commitment to support α from a particular task (i.e. $DT_\alpha \neq \varnothing$), then no such resolver is applicable. Indeed, any supporter would appear by a mean of task decomposition only.

  In the other case (i.e. $DT_\alpha = \varnothing$), we need to consider the insertion of free actions outside of any existing decomposition trees. Let AS_α be a set of action templates A such that $\langle sv{=}v \rangle \in E^+_A$. For any action $A \in AS_\alpha$, a possible resolver for α is to insert an instance a of A and add $task(a)$ to DT_α. This resolver ensures that α will

be supported either by a direct effect of a or by an effect of one of its subactions. The choice of the actual supporting assertion for α is delayed.

To summarize, our definition of unsupported assertions requires every assertion to have a causal supporter whose choice is made explicit by the introduction of a causal link. We distinguish three cases for the selection of the causal supporter depending on whether it is already in the plan, can be introduced by decomposing existing tasks or requires the introduction of new actions outside of existing decomposition tree. In the two latter choices, the selection of the actual causal supporter is delayed to a later time where the task network has seen more decompositions.

3.2.2.2 Unrefined tasks

We say that a task is *unrefined* if it has been required, either as an initial task of the problem or as subtask of an action in the plan, and it has not been refined by a *task decomposition* transformation (Section 2.5.1). Given a partial plan $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$, any task $\tau \in \pi_\phi$ is an unrefined task. Resolving an unrefined task requires selecting an action that achieves it and applying the corresponding task decomposition transformation.

Given an unrefined task $\tau : \texttt{tsk}(x_1, \ldots, x_n)$, let PR_τ be the set of action templates A such that $task(A)$ is unifiable with τ (i.e. $task(A)$ is of the form $\texttt{tsk}(b_1, \ldots, b_n)$). For any action template $A \in PR_\tau$, a possible resolver to the unrefined task τ is to apply the decomposition transformation $\phi \xrightarrow{\tau, a}_D \phi'$, where a is an instance of A. Thus, in the resulting chronicle ϕ' the unrefined task τ has been replaced by a new action instance a.

Furthermore, we should account for the commitment already made for the support of unsupported assertions. For this reason, we only consider as a resolver an action template $A \in PR_\tau$ if for any unsupported assertion α such that $\tau \in DT_\alpha$, A has a possible effect $e \in E^+_\tau$ that could support α.

In practice, this definition is close to the decomposition procedure of HTN planning: for each pending task, a possible choice is to apply one of the actions achieving it. Our procedure simply extends it to account for the commitments previously made while resolving unsupported assertion flaws. An example of the interactions that can result from that are given in Example 3.2.

3.2.2.3 Conflicting assertions

Our last type of flaw occurs when the chronicle $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$ has a possible instantiation that would make two assertions conflicting. We start by studying the simple case of persistence assertions and then show how this scheme can be used to consider the conflicts involving change assertions as well.

Two persistence assertions $p_1 = \langle [t^s_1, t^e_1]\, sv_1 = v_1 \rangle$ and $p_2 = \langle [t^s_2, t^e_2]\, sv_2 = v_2 \rangle$ are conflicting when they can concurrently require different values for the same state variable. More precisely they are seen as conflicting when the three following conditions are met:

- they can temporally overlap, meaning that the temporal constraints in $\mathcal{C}_\phi$ are consistent with $[t^s_1, t^e_1] \cap [t^s_2, t^e_2] \neq \varnothing$.
- they can be about the same state variable. This is the case when sv_1 and sv_2 are respectively of the form $\texttt{sv}(x_1, \ldots, x_n)$ and $\texttt{sv}(y_1, \ldots, y_n)$ and $\mathcal{C}_\phi \cup \{ x_1 = y_1 \wedge \cdots \wedge x_n = y_n \}$ is a consistent set of constraints.

- their values can be different, i.e., $v_1 \neq v_2$ is consistent with $\mathcal{C}_\phi$.

Resolving a conflict requires enforcing that the two assertions can not be conflicting in any of the instantiation of the chronicle. A conflict involving a pair of assertions (p_1, p_2) can be resolved by a plan restriction $\phi \xrightarrow{(\varnothing, \{c\})}_R \phi'$ where c is one of:

- a temporal separation constraint forcing p_1 and p_2 to be non-overlapping, i.e., c is either $end(p_1) < start(p_2)$ or $end(p_2) < start(p_1)$
- a state variable separation constraint, i.e., c is one of $\{ x_1 \neq y_1, \ldots, x_n \neq y_n \}$ where $x_{1..n}$ and $y_{1..n}$ denote the arguments of the state variables of p_1 and p_2 respectively.
- a unification constraint that requires the two values v_1 and v_2 to be the same $(v_1 = v_2)$.

A change assertion $[t_s, t_e]\, sv : v_s \mapsto v_e$ is more complex and, for the purpose of identifying conflicts can be seen as a combination of three persistences:

- a start persistence assertion $[t_s]\, sv = v_s$
- a changing part $[t_s + 1, t_e - 1]\, sv = \text{UNDEFINED}$, where UNDEFINED is a special value that is not unifiable with any variable including UNDEFINED itself (i.e. $\forall v, \text{UNDEFINED} \neq v$).
- an end persistence $[t_e]\, sv = v_e$

A conflict involving a change assertion can be identified, and resolved, by reasoning on its three component as it is done for persistences. We note that this definition allows the start and end instants of the change assertion to overlap with other assertions, given that they require the same value. However, the inner part of the change assertion that denotes the interval over which the value is changing can not overlap with any other assertion about the same state variable.

In practice, conflicting assertions are simply an extension of the notion of threats in plan space planning to a context of partially instantiated partial plans with assertions spanning over temporal intervals. The resolvers thus consider separation constraints not only on the temporal aspects but also on the different variables involved in the two threatening assertions.

Example 3.2. Let us consider the following scenario with an initial chronicle

$$\phi_0 = (\{ t : \texttt{go}(\texttt{r1}, \texttt{d1}) \}, \{ [0]\ \texttt{loc}(\texttt{r1}) := \texttt{d2}, [100, 110]\ \texttt{loc}(\texttt{r1}) = \texttt{d1} \}, \mathcal{C}_0)$$

where t is an unrefined task requiring `r1` to go to `d1` and the second persistence in ϕ_0 requires `r1` to be in `d1` during $[100, 110]$. $\langle [0]\ \texttt{loc}(\texttt{r1}) := \texttt{d2} \rangle$ gives the initial position of `r1` and is a priori supported. Our model is fully hierarchical and contains two task dependent actions achieving the task `go`. The first one is the phantom action that just requires r to be in the right location already and the second is the action of actually moving. While somewhat artificial in this simplistic example, the phantom action is usually needed in hierarchical problems in which goals are stated as tasks.

no-move(r,d')	
task:	go(r,d')
dependent:	yes
assertions:	$[t_{start}, t_{end}]\, loc(r) = d'$
constraints:	$\varnothing$

move(r,d,d')	
task:	go(r,d')
dependent:	yes
assertions:	$[t_{start}, t_{end}]\, loc(r) : d \mapsto d'$
constraints:	$t_{end} - t_{start} = 40$

This chronicle contains two flaws: an unrefined task and an unsupported assertion. We simulate the behavior of a planner that would try to solve this simple planning problem. We distinguish two scenarios, each representing a preference of the planner as of which type of flaw to handle first.

Scenario 1. We first choose to resolve the unrefined task t. This task has two actions possibly refining it, resulting in two resolvers:

- we can choose to refine t with no-move, resulting in a chronicle

$$\begin{aligned}\phi_1 = (\{\, & a_1 : \mathsf{no\text{-}move}(\mathtt{r1}, \mathtt{d1})\,\},\\ & \{\,[0]\ \mathit{loc}(\mathtt{r1}) := \mathtt{d2},\ [start(a_1), end(a_1)]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1},\\ & \ [100, 110]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1}\,\},\\ & \mathcal{C}_1)\end{aligned}$$

  The resulting chronicle has two unsupported assertions. Note that there is no existing change assertion (or a priori supported assertion) that can support the claim that r1 is in d1. Furthermore, no additional action can be added to ϕ_1 because all actions are task-dependent and there are no unrefined tasks left. Thus, none of the flaws have resolvers and ϕ_1 is a dead-end from which the planner will need to backtrack.

- or, we can choose to refine t with move, resulting in the chronicle

$$\begin{aligned}\phi_2 = (\{\, & a_2 : \mathsf{move}(\mathtt{r1}, d, \mathtt{d1})\,\},\\ & \{\,[0]\ \mathit{loc}(\mathtt{r1}) := \mathtt{d2},\ [start(a_2), end(a_2)]\ \mathit{loc}(\mathtt{r1}) : d \mapsto \mathtt{d1},\\ & \ [100, 110]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1}\,\}, \mathcal{C}_2)\end{aligned}$$

  Here we also have two unsupported assertions to support and no action can be added to the plan. Let us suppose we choose the first one,$\langle [start(a_2), end(a_2)]\ \mathit{loc}(\mathtt{r1}) : d \mapsto \mathtt{d1}\rangle$, as the next flaw to handle. Our only resolver is to select $\langle [0]\ \mathit{loc}(\mathtt{r1}) := \mathtt{d2}\rangle$ as the supporting assertion and enforce its support by the addition of a causal link $\langle [0, start(a_2)]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d2}\rangle$ and binding d to d2.

$$\begin{aligned}\phi_3 = (\{\, & a_2 : \mathsf{move}(\mathtt{r1}, \mathtt{d2}, \mathtt{d1})\,\},\\ & \{\,[0]\ \mathit{loc}(\mathtt{r1}) := \mathtt{d2},\ [0, start(a_2)]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d2},\\ & \ [start(a_2), end(a_2)]\ \mathit{loc}(\mathtt{r1}) : \mathtt{d2} \mapsto \mathtt{d1},\ [100, 110]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1}\,\}, \mathcal{C}_3)\end{aligned}$$

The only flaw left in ϕ_3 is the unsupported goal assertion that $\texttt{loc}(\texttt{r1})$ be in $\texttt{d1}$ at time 100. The only supporter for this flaw is to take as the supporting assertion the change from $\texttt{d2}$ to $\texttt{d1}$, $\langle [start(a_2), end(a_2)]\ \texttt{loc}(\texttt{r1}) : \texttt{d2} \mapsto \texttt{d1} \rangle$, enforce the support by the introduction of the persistence $\langle [end(a_2), 100]\ \texttt{loc}(\texttt{r1}) = \texttt{d1} \rangle$. This will result in a solution plan:

$$\begin{aligned}\phi^* = (&\{\, a_2 : \mathsf{move}(\texttt{r1}, \texttt{d2}, \texttt{d1}) \,\},\\ &\{\, [0]\ \texttt{loc}(\texttt{r1}) := \texttt{d2},\ [0, start(a_2)]\ \texttt{loc}(\texttt{r1}) = \texttt{d2},\\ &\quad [start(a_2), end(a_2)]\ \texttt{loc}(\texttt{r1}) : \texttt{d2} \mapsto \texttt{d1},\\ &\quad [end(a_2), 100]\ \texttt{loc}(\texttt{r1}) = \texttt{d1},\ [100, 110]\ \texttt{loc}(\texttt{r1}) = \texttt{d1} \,\}, \mathcal{C}^*)\end{aligned}$$

Scenario 2. Suppose now that we were to choose the unsupported assertion $g = \langle [100, 110]\ \texttt{loc}(\texttt{r1}) = \texttt{d1} \rangle$ as the first flaw to solve. We first note that there is no assertions already in ϕ_0 that could be used as a direct supporter for g. Also, all our actions are *task dependent*, thus we have no possibilities of inserting a *free* action that would provide the desired effect. Our only possibility is thus to choose a task that has $\langle \texttt{loc}(\texttt{r1}) = \texttt{d1} \rangle$ in its possible effects. The only such task is $t : \texttt{go}(\texttt{r1}, \texttt{d1})$, the flaw thus have a single resolver. Applying this resolver does not result in any modification of chronicle, instead the planner will simply record that the supporter of g must be chosen in the assertions appearing in refinements of t. This is done by adding t to DT_g and postponing the actual resolution of g.

The only active flaw left is the unrefined task t. Like in the first scenario, we have two candidates for refining it. Moreover, we now need to ensure that any refinement we make can eventually provide a supporter for g. The first action no-move, has no possible effect. Hence it cannot be used to refine t, as it would violate our previous commitment to choose a supporter for g in the descendants of t. Our only choice is to refine t with $\mathsf{move}(\texttt{r1}, d, \texttt{d1})$ that has $\langle \texttt{loc}(\texttt{r1}) = \texttt{d1} \rangle$ in its possible effects.

$$\begin{aligned}\phi_4 = (&\{\, a_3 : \mathsf{move}(\texttt{r1}, d, \texttt{d1}) \,\},\\ &\{\, [0]\ \texttt{loc}(\texttt{r1}) := \texttt{d2},\\ &\quad [start(a_3), end(a_3)]\ \texttt{loc}(\texttt{r1}) : d \mapsto \texttt{d1},\\ &\quad [100, 110]\ \texttt{loc}(\texttt{r1}) = \texttt{d1} \,\},\\ &\mathcal{C}_4)\end{aligned}$$

Now that t has been refined, the unsupported assertion $g = [100, 110]\ \texttt{loc}(\texttt{r1}) = \texttt{d1}$ can be reconsidered. Its only possible resolver is to select the newly inserted statement $\langle [start(a_3), end(a_3)]\ \texttt{loc}(\texttt{r1}) : d \mapsto \texttt{d1} \rangle$ as the supporting assertion. This is a valid resolver as it was inserted by an action deriving from the task t. As in the previous scenario, the insertion of the causal support constraints for both unsupported

assertions would result in a solution:

$$\begin{aligned}\phi^{*\prime} = (\{\, & a_3 : \mathsf{move}(\mathtt{r1}, \mathtt{d2}, \mathtt{d1})\,\},\\ & \{\,[0]\ \mathit{loc}(\mathtt{r1}) := \mathtt{d2},\ [0, \mathit{start}(a_3)]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d2},\\ & \quad [\mathit{start}(a_3), \mathit{end}(a_3)]\ \mathit{loc}(\mathtt{r1}) : \mathtt{d2} \mapsto \mathtt{d1},\\ & \quad [\mathit{end}(a_3), 100]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1},\ [100, 110]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1}\,\}, \mathcal{C}^{*\prime})\end{aligned}$$

We can see that both scenarios do find the same solution, the only difference being the name given to action instances (a_2 vs a_3). $\phi^{*\prime}$ is equivalent to the solution found in the first scenario, by means of a simple renaming of the action instance. An overview of the solution is given in Figure 3.1. The solution would notably contains the following temporal constraints.

$$\begin{aligned}\mathcal{C}^{*\prime} = \{\, & \mathit{start}(t) = \mathit{start}(a_3) \wedge \mathit{end}(t) = \mathit{end}(a_3) \wedge 0 \leq \mathit{start}(a_3)\\ & \wedge \mathit{end}(a_3) \leq 100 \wedge \mathit{end}(a_3) - \mathit{start}(a_3) = 40 \ldots\,\}\end{aligned}$$

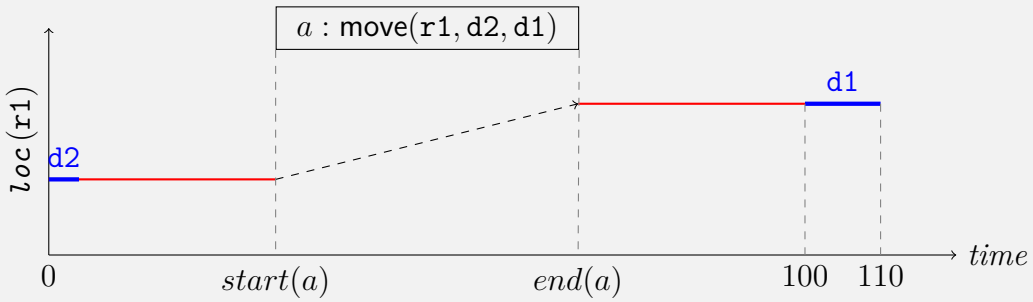


Figure 3.1: Solution to the planning problem. In blue are the two assertions part of initial problem definition. In red are the persistence assertions inserted to enforce causal support.

3.2.3 Constraint Networks

FAPE makes an important use of constraint networks to represent temporal aspects of the problem and to maintain partially instantiated partial plans. The constraint networks are meant to efficiently store the constraints accumulated by the planner and answer various queries necessary for the identification of flaws as well as the listing of possible resolvers.

For instance determining whether two assertions are conflicting requires determining if they can temporally overlap and if all variables appearing in the state variable are unifiable. For this purpose, we have two distinct constraint networks: one responsible for temporal reasoning and one for handling binding constraints. Some variables are present in both constraints networks: duration variables, such as δ in Figure 2.1, that need to meet binding as well as temporal constraints. We start by presenting the temporal and binding constraint networks independently, and will then show how they are extended to handle constraints involving duration variables.

3.2.3.1 Temporal Constraints Network

Temporal variables (or timepoints) and temporal constraints are handled as a Simple Temporal Problem (STP) [DMP91]. Given a set of timepoints $\mathcal{X}$, a temporal constraint is an inequality of the form $t_2 - t_1 \geq d$ where t_1 and t_2 are timepoints in $\mathcal{X}$ and d is an integer constant. Such a constraint means that the minimal delay from t_1 to t_2 must be of at least d. The use of a Simple Temporal Network (STN) answers some efficiency requirements as both the consistency and the minimal network of an STN can be computed in polynomial time. On the down side, its restriction to only represent a conjunction of constraints forces the planner to branch on some scheduling choices that could have been encoded has a Disjunctive Simple Temporal Network (DSTN) or as a Temporal Constraint Satisfaction Problem (TCSP). However, determining the consistency of such temporal constraint network is NP-complete [DMP91].

Propagation uses Floyd-Warshall's All-Pairs Shorter Path Algorithm to get the minimal network [Flo62; War62]. In practice, this means that in a fully propagated network, the minimal delay between all pairs of timepoints is known and explicitly represented. For a network containing n timepoints, the cost of this operation is $\Theta(n^3)$ for any network. We use the incremental version of this algorithm, as presented by Planken [Pla08], that has a quadratic runtime complexity of $O(n^2)$. Having the minimal network allows us to answer all temporal queries in constant time

While many other more efficient techniques have been devised to determine the consistency of an STN (e.g. [CO96b; XC03; PWY10; TPW11; PWK08]), they are mainly based on avoiding the propagation of constraints not needed for determining consistency of the network. As a result, answering some queries would require additional computations, often taking the form of a Bellman-Ford algorithm to find the minimal distance between two timepoints.

We denote as $dist_{STN}(t_1, t_2)$ the minimal delay that must elapse between t_1 and t_2. It can be extracted in constant time from the minimal network. We further note as $\mathcal{O}$ the timepoint denoting the origin of time.

3.2.3.2 Binding Constraints Network

A binding constraint network is a tuple $(\mathcal{X}_{Bind}, \mathcal{D}_{Bind}, \mathcal{C}_{Bind})$ where:

- $\mathcal{X}_{Bind}$ is a set of object variables and duration variables.
- $\mathcal{D}_{Bind}$ is a set of domains for each variable in $\mathcal{X}_{Bind}$. The initial domain of object variables is a subset of the objects in the problem determined by their types. The initial domain of duration variables is the set of integers denoting durations that appear in the problem definition.
- $\mathcal{C}_{Bind}$ is a set of constraints on the variables of $\mathcal{X}_{Bind}$. A constraint is either
  - an equality constraint, e.g., $x_1 = x_2$
  - a difference constraints, e.g., $x_1 \neq x_2$
  - a general relation constraint, e.g., $\textit{travel-time}(x_1, \ldots, x_{n-1}) = x_n$. Relation constraints are associated with a relation table (e.g. $\gamma_{\textit{travel-time}}$ of Table 2.1) that give the allowed tuple of values for the variables $x_{1..n}$.
  - an inequality on a duration variable, e.g., $x_1 \leq 12$ where x_1 is a duration variable.

– disjunctive equality constraint, e.g., $x = x_1 \vee x = x_2 \vee \cdots \vee x = x_n$

The main purpose of the constraint network is to detect inconsistencies in the set of constraints and to answer various queries on binding variables. Typical queries on the constraint network include *(i)* the domain of a variable, *(ii)* knowing if two variables are equal, *(iii)* knowing if two variables can be made equal or different. Answering those queries exactly is far from trivial as many constraint satisfaction problems are NP-complete. In particular, the only presence of inequality constraints is sufficient to demonstrate the membership in NP.

To find a trade-off between accuracy and efficiency we use an arc-consistency algorithm (AC-3 [Mac77]) for constraint propagation. More specifically, we maintain a work list of constraints $\mathcal{Q}_{Bind}$. When a new constraint is added to the network, it is added to $\mathcal{Q}_{Bind}$. Propagation proceeds until $\mathcal{Q}_{Bind}$ is empty as follows:

1. extract a constraint c from $\mathcal{Q}_{Bind}$,

2. for every variable x involved in c, remove any value v from $dom(x)$ that can not satisfy this constraint.

   For a relation constraint $c = \langle R(x_1, \ldots, x_{n-1}) = x_n \rangle$, this is done by first eliminating from the relation any tuple of values $(v_1, \ldots, v_n)$ that is not achievable (i.e. $\exists i \in [1, n] \,|\, v_i \notin dom(x_i)$). For all remaining tuples, projecting them on a single variable x_i gives a set of allowed values $Proj^c_{x_i}$. The domain of each variable x_i is restricted to the values in $Proj^c_{x_i}$.

   For a disjunctive equality constraint $c = \langle x = x_1 \vee x = x_2 \vee \cdots \vee x = x_n \rangle$, the domain of x is restricted to $\bigcup_{i \in [i,n]} dom(x_i)$.

3. if the domain of any variable x was updated during propagation, all constraints involving x are added to $\mathcal{Q}_{Bind}$. If $\mathcal{Q}_{Bind}$ is not empty, then go back to step 1.

If a variable $x \in \mathcal{X}_{Bind}$ has an empty domain, then the network is not consistent. Otherwise the network is only *arc-consistent* which is used as an optimistic approximation of consistency by the planner. Full consistency is only checked to ensure that a plan with no flaw is indeed a solution to a planning problem (i.e. that the binding constraint network is indeed consistent).

The network is used as a base for various queries from the planner. Those are used to identify flaws and filter out impossible resolvers. The binding constraint network considers that two variables x_1 and x_2 are:

- *unified* if they have the same singleton domain or if there is an equality constraint between the two. This equality constraint can involve intermediate variables.

- *unifiable* if they have intersecting domains and there is no inequality constraints between them (or between any two variables unified with x_1 and x_2 respectively).

- *separated* if they are not unifiable. That is, if they have non intersecting domains or if there is an inequality constraint between them (or between any two variable unified with them).

- *separable* if they are not unified.

3.2.3.3 Duration Constraints

A special case of variables are duration variables that appear both as variables in the binding constraint network and in duration constraints in the temporal network. A duration constraint is thus a special case of a mixed constraint of the form $t_2 - t_1 \geq f(x)$ where:

- t_1 and t_2 are timepoints of the temporal network,
- x is a duration variable in the binding constraint network,
- $f : \mathbb{Z} \to \mathbb{Z}$ is a function transforming the value of x into the one appearing in the temporal constraint

For instance, the introduction of an instance a of the move action of Figure 2.1, would result in the following constraints:

$$\begin{aligned} &\mathit{travel\text{-}time}(r, d, d') = \delta \\ \wedge\ &\mathit{end}(a) - \mathit{start}(a) \geq \delta \\ \wedge\ &\mathit{start}(a) - \mathit{end}(a) \geq -\delta \end{aligned}$$

where r, d and d' are object variables, $\mathit{start}(a)$ and $\mathit{end}(a)$ are timepoints and δ is a duration variable. The last two constraints are mixed constraints as they involve variables from the temporal network and from the binding constraint network.

The propagation of mixed constraints requires communication between temporal and binding constraint networks. This is done through the addition of new constraints in a network when a new bound is found on the value of x in the other network. More specifically, given a mixed constraint $t_2 - t_1 \geq f(x)$:

- when a new constraint $d \geq t_2 - t_1$ is added or inferred in the temporal network, the domain of x is filtered to remove any value that would be greater than d after transformation by f:

  $$\mathit{dom}(x) \leftarrow \{\, v \mid v \in \mathit{dom}(x) \wedge d \geq f(v) \,\}$$

  If the domain of x is modified by this operation, it will trigger propagation in the binding network.

- when the domain of the duration variable x is modified in the binding constraint network, then a new temporal constraint $t_2 - t_1 \geq \min_{v \in \mathit{dom}(x)} f(v)$ is added to the STN. In practice, this means that we propagate the least constraining instantiation of x into the temporal network.

3.2.4 Search Space: Properties and Exploration

We now give some characteristics of the search space.

Search Space Infiniteness. In plan-space planning, the search space is infinite which can be easily understood as the planner explore the set of plans which is infinite [GNT04, Chap. 5]. Indeed, even for a simplistic goal such as going somewhere, there can be an infinite set of plans fulfilling it: one can go round and round in circles an arbitrary number of time before getting to his destination. This in an important difference with state-space classical planners that prohibit going through the same state twice. The set of states being finite, their search space is finite as well. The search space of HTN planners is infinite because of the presence of recursive methods [EHN94]. Since our search procedure is an extension of plan-space planning (and can represent HTN problems) our search-space is infinite as well. In these conditions, the planner must guarantee a methodical exploration of the search space to ensure that, if there is a solution plan, it will be found. Algorithms such as depth bounded search or incremental deepening meet those expectations. More interestingly, best first search in general (and A* in particular) also meet those expectations as long as the addition of an action to the partial plan constitute a strictly positive cost for its evaluation function. Indeed when the number of actions in a plan augments towards infinity, the evaluation functions will eventually prefer a shallower plan that is shorter. Our planner uses a best-first search algorithm whose evaluation function allows for a methodical exploration (as we will detail in Section 3.3.4.1).

Search Space Acyclicity Another condition for the completeness of the search is the acyclicity of the search space, meaning that the search space does not contain any loop that would allow indefinitely coming back to a given partial plan.

Proposition 3.2.1. *The search space of* FapePlan *is acyclic.*

Proof. To show the acyclicity of the search space, we show that given a partial plan, there is no infinite sequence of resolver application that can lead to the same partial plan.

We first observe that the Transform(ϕ, ρ) of Algorithm 1 is incremental: it can add actions or constraints but never removes anything. Furthermore, each Transform application will result in one flaw being solved (even in the case where the resolution of an unsupported assertion is delayed, this unsupported assertion will only be reconsidered once at least one other flaw has been fixed). Since there is only a finite number of flaws in a partial plan, the repeated application of transform will either *(i)* result in a solution plan with no flaw, *(ii)* result in an inconsistent plan or *(iii)* add a new action to the plan, possibly introducing new flaws. In the latter case, the added action can never be removed and the planner cannot transform it back into the original chronicle. □

Soundness & Completeness. We now study the soundness and completeness of our planning procedure.

Proposition 3.2.2. FapePlan *(Algorithm 1) is sound.*

Proof. Soundness requires proving that any plan returned by FapePlan is indeed a solution, i.e., that it respects the four conditions of Definition 2.5.2.

The first one, reachability of ϕ^* from ϕ_0, can be shown since *(i)* any transformation made to the plan is done by applying a resolver, and *(ii)* all resolvers can be expressed in terms of the allowed transformations (task decomposition, free action insertion and restriction insertion). As reachability admits any sequence of transformations, the application of any number of resolvers, regardless of their order will result in a plan that is reachable from the original one.

The last three conditions all correspond to a flaw type. Hence, if any of those conditions is not met, the partial plan would have a flaw that the planner will need to resolve. The requirement that the plan be flaw free, together with the type of flaws considered is thus sufficient to guarantee that the last three conditions are met. □

Proposition 3.2.3. FapePlan *(Algorithm 1) is complete.*

Proof. We rely on the study by Kambhampati, Knoblock, and Yang [KKY95] and Schattenberg [Sch09, Sec. 2.6 to 2.8] of the general refinement planning procedure where a set of *deficiency detection functions* identify flaws in a partial plan and a set of *modification generation functions* generate modifications of the plan that fix the flaws (i.e. resolvers). Our procedure is an instantiation of this more general scheme with three detection functions (one for each flaw) and their modification generation functions implicitly defined by the set of resolver associated to a flaw. Showing completeness of a particular refinement planning procedure requires us to show that *(i)* no solution plan is rejected because it has flaws, *(ii)* for a given flaw, our resolvers cover all the ways of addressing it [Sch09, Def 3.2].

Let us first show that no solution plan is ruled out because of the presence of flaw. We assume that a solution plan ϕ has a flaw of a given type and show that it cannot be a solution plan or that it will be transformed in a flawless solution plan.

- if ϕ has an unrefined task τ then it is not a solution plan according to Definition 2.5.2.

- if ϕ is detected with an unsupported assertion α, it means that we have not explicitly added a causal link from a supporting assertion β. Assuming ϕ to be solution, it means that α is supported by the presence of chain of assertions that must eventually originate in a change assertion or in an a priori supported assertion β. β is thus a causal support of α and there is a chain of persistences preventing any change on their state variable during $[end(\beta), start(\alpha)]$. Even though this situation could trigger a flaw, its resolver would be a causal link $\beta \rightarrow \alpha$ that would simply make the support explicit in an equivalent plan.

- Similarly, the planner could detect a conflict between two assertions that cannot be conflicting due to implicit constraints. Indeed, the use of arc-consistency in place of full consistency in the binding constraint network could make FAPE miss such implicit constraints. Here again, the planner would simply provide a resolver making the constraint explicit and resulting in an equivalent plan.

We now show that no solution plan is missed due to an incomplete set of resolvers:

- **Unsupported Assertion.** It is easy to see, by the need of causal support, that all assertions in a solution plan must have an incoming causal link from a supporting assertion. For completeness, we need to show that all possible supporters are considered regardless of whether they are in the current partial plan or can be inserted later. Our approach distinguishes the assertions already present in the plan from those that will be inserted later. The *direct supporter* resolver provides the possibility for the planner to select any assertion already in the plan.

  Regarding the assertions not yet in the plan we distinguish two cases: the supporter can be introduced by refining an existing task or it can be introduced by an action

not descending from an existing task. The former is handled by the *delayed support from existing task* resolvers that allows choosing any of the future assertions deriving from an existing task. For the latter, let us observe that the containing action will be part of another decomposition tree not yet in the plan, whose root is necessarily a free action. Our *delayed support from new free action* resolvers allow precisely to consider the addition of free actions as the source of new supporting assertion, regardless of whether they appear in the free action it self or in a descendant action obtained by decomposing its subtasks.

To summarize our resolvers allow to consider all possible sources of supporting assertions, namely: *(i)* those already in the plan, *(ii)* those introduced by the extension of existing decomposition trees, and *(iii)* those introduced by the expansion of new decomposition trees.

- **Unrefined task.** Assuming that no previous commitment was made on the support of unsupported assertions, the set of resolvers is complete as it considers all possible task decomposition transformations. Let us assume now that the planner made an earlier choice regarding the support of an assertion α: it decided that α must be supported by a descendant of an unrefined task τ (i.e. $\tau \in DT_\alpha$). This can lead the planner to disregard an action template A for the refinement of τ because A would not have any possible effect for supporting α. This pruning is however sound because there cannot be an assertion supporting α deriving from A, i.e., there is no solution plan involving the decomposition of τ with A given our previous commitment. Furthermore, possibility of using A to refine τ will be considered in other branches of the search space: those deriving from another resolver for α.

- **Conflicting assertions.** For two assertions α_1 and α_2 to be conflicting in a partial plan $(\pi, \mathcal{F}, \mathcal{C})$, it is necessary that a conjunction of constraints be entailed by $\mathcal{C}$: *(i)* they overlap (i.e. $end(\alpha_1) \geq start(\alpha_2) \wedge end(\alpha_2) \geq start(\alpha_1)$), *(ii)* all the arguments of their state variables are equal, and *(iii)* in the case of two persistence assertions their values are different. The negation of this conjunction of constraints thus forms a disjunction of constraints that must hold in any solution plan (otherwise, α_1 and α_2 would be conflicting). Each of the disjunct corresponds to a resolver of the threat. Because at least one such disjunct must hold in a solution plan, the set of resolvers for conflicting assertions is complete.

□

Redundancy in the Search Space (Systematicity). Another desirable property is the absence of redundancy in the search space, denoting the fact that there are no two partial plans that can be refined into two equivalent plans. This property is usually referred to as *systematicity* of the planning algorithm [MR91; Kam93]. In practice, it means that, for a given order in which flaws are resolved, there is a single path from the initial plan to any partial plan. The key idea for achieving systematicity in plan-space planning is to make sure that all resolvers of a given flaw are mutually exclusive [Kam93].

Mutual exclusivity between resolvers is easily achieved by totally instantiated plan-space planners such as UCPOP [PW92], e.g., a threat can only be resolved by a strictly before or strictly after temporal constraint. Our least-commitment approach with partially instantiated partial plans makes the achievement of systematicity harder. In order to achieve systematicity, McAllester and Rosenblitt [MR91] extend the lifted plan-space

planning algorithm as follows: whenever the planner needs to know whether two variables v_1 and v_2 are equal, two mutually exclusive branches of the search space are generated: one with the constraint $v_1 = v_2$ and one with the constraint $v_1 \neq v_2$. This mechanism is shown to achieve systematicity but results in an early commitment that is detrimental to search efficiency [Kam93]. Indeed, as noted by Kambhampati [Kam93], removing all redundancy usually requires an early commitment that has a negative impact on the compactness of the search space, each partially instantiated plan representing less fully instantiated ones.

Our approach makes a trade-off between systematicity and commitment. Our use of a (single sourced) causal link achieves systematicity as it makes a commitment to a single achiever [Tat76; MR91]. Our resolvers for conflicting assertions however permit redundancy in the search space as they are not all mutually exclusive. A practical technique to achieve systematicity in many planning problems is to delay the handling of threats. Indeed, achieving complete causal support usually results in a complete instantiation of the plan. If a threat appears in a totally instantiated plan, the only possible resolvers are (mutually exclusive) temporal separation constraints. This technique to improve systematicity is used in our preferences for flaw selection (see Section 3.3.4.1).

HTN planners generally do not achieve systematicity but instead can consider a plan as many times as there are ways to decompose it from the initial task network. It is the responsibility of the domain designer to come up with domains where the planner has no or few redundancies in the search space. Planners like SHOP or UMCP only consider solution plans composed of primitive actions. Because in our model, all actions (including high level actions) appear in the solution plan, our solution contains a memory of the path taken to reach it. Systematicity is not directly comparable with the one off those HTN planners. We can simply state that, if one is only interested in the primitive actions of a plan (e.g. would post-process our plan to delete all high-level actions) then our procedure lacks systematicity in a similar fashion as other HTN planners. Forward search HTN planners, such as SHOP2 or SIADEX [Nau+03] have another source of redundancy in their search space has they branch on the order in which to decompose tasks. We discuss this limitation, and how it is avoided in FAPE, in details in Section 3.4.2.

Even when one is interested in all high-level actions of a plan, our resolvers for unrefined tasks do not achieve systematicity. As a counter example, suppose that the solution plan contains two distinct actions achieving the same task symbol over the same temporal interval. This solution plan can appear twice in the search space as two identical tasks could interchangeably be achieved by any of those actions. In practice however, such cases are rare and the lack of systematicity of the hierarchical part of the planning process can be ignored like in other hierarchical planners.

Discussion. Our definition of flaws and resolvers results in a top down approach of hierarchical planning, similar to that other HTN planner such as SHOP. Indeed, the search starts from an existing task network that is entirely decomposed over the course of planning. In addition, we permit the insertion of free action to be added independently of the existing decomposition trees.

In practice, this means that any action in the plan was inserted later than all actions it descends from. This scheme has the advantage of being well documented and understood as it is the technique used by all HTN planners to date. This top down decomposition schema also has the advantage of mimicking some of the reasoning process followed by humans which makes the understanding of the planning process easier.

Other techniques are however possible. For instance, one could allow the direct insertion of task-dependent actions that are needed for their effects and then build the plan in a bottom up fashion until meeting the initial task network. In this setting, we would require an additional flaw tracking whether a task-dependent action a has been attached to a task. Solving this flaw would involve either marking a as refining an existing task τ or adding an high level action with a subtask that can be refined with a. While we tested some of those techniques in FAPE, we do not report on them in this chapter as they resulted in poor performance in a pure task planning context. Indeed, this scheme complicates the definition of dedicated heuristics since one must not only reason on what are the effects achievable through the decomposition of unrefined task but also on how actions already in the plan can be linked to the existing task network. Those techniques can however be of interest in contexts relevant to our work such as plan repair and we will study some of those in Section 5.5.4.3.

3.3 Search Control

Section 3.2.4 has defined the search space of FAPE, obtained by extending the PSP algorithm with hierarchical considerations. Compared to most planners, the search space of constraint-based planners (including FAPE) is much more compact as their lifted representation allow representing many partial plan in a single search node. On the other hand, the maintenance of various constraint networks make the expansion of a search node computationally expensive and such planners still require search control strategies to perform efficiently.

In the following sections, we detail the different strategies used for the shaping and exploring of the search space. These can be roughly separated in three categories:

- the strategy for *choosing which flaw to solve next* plays an important role in plan-space planning as it allows shaping the search space. For instance, choosing to first decompose all unrefined tasks would lead to very different search space than the ones resulting in solving threats first.

- the strategy of *which partial plan to consider next* is critical to guide the exploration of the search space towards solution plan, even for planning problems involving only a handful of actions.

- *inference of necessary constraints* to allow an early detection of specific characteristics of a partial plan. An example of such inference capability is the detection that a given unsupported assertion cannot be achieved early because its establishment requires a long chain of actions. Such new constraints can be used to reduce the search space by detecting dead-ends or filtering out impossible resolvers.

These three aspects are strongly interconnected since flaw ordering and inference can be seen as shaping up the search space that will be explored given the partial plan selection strategy. This section will first study the techniques used for automatic inference in FAPE (Sections 3.3.1 to 3.3.3). More specifically, Section 3.3.2 will detail a reachability analysis to infer what cannot be achieved from a partial plan due to its temporal and hierarchical features and Section 3.3.3 will provide techniques to reason on the causal structure of a plan. Finally, Section 3.3.4 will focus on the strategies for flaw and plan selection.

3.3.1 Instantiation and Refinement Variables

While the lifted representation is beneficial for the efficiency of the search, it makes reasoning on a partial plan harder because a given lifted statement can be refined into many ground ones. This can be detrimental as most heuristic computation for planning require a ground representation. To facilitate the mapping between lifted partial plans and ground heuristic techniques, we introduce two types of variables that represent the possible instantiation of actions and the set of ground actions that can be used to refine a task.

Any action template act is associated with a relation $\gamma_{\text{act-inst}}$ that contains all possible ground instances of act. A ground instance of act is one where all object variables are bound and satisfy all binding constraints in the template. If act has a ground instance $id = \langle \mathsf{act}(\mathtt{c1}, \ldots, \mathtt{cn}) \rangle$, then $\gamma_{\text{act-inst}}$ will contain the tuple $(\mathtt{c1}, \ldots, \mathtt{cn}, id)$ where id is a unique identifier of this instance. Table 3.1 gives an example of this relation for the move action of Figure 2.1.

Robot	Origin	Destination	Action ID
r1	d1	d2	id_1
r1	d2	d3	id_2
r1	d2	d1	id_3
r2	d1	d2	id_4
r2	d2	d3	id_5
	...		

Table 3.1: Table for the relation $\gamma_{\text{move-inst}}$ that give the instantiations of the move action. In this example, id_1 identifies the ground action $\mathsf{move}(\mathtt{r1}, \mathtt{d1}, \mathtt{d2})$.

A lifted action $a : \mathsf{act}(x_1, \ldots, x_n)$ in π_ϕ is associated with an instantiation variable I_a that takes a value in the set of ground instances of act. At a given time, $dom(I_a)$ should contain any ground action that a might become once all its parameters are instantiated:

$$\mathsf{act}(\mathtt{c1}, \ldots, \mathtt{cn}) \in dom(I_a) \iff \forall_{i \in 1..n}\ \mathtt{ci} \in dom(x_i)$$

To enforce, this relationship, every time a (lifted) action instance $a = \langle \mathsf{act}(x_1, \ldots, x_n) \rangle$ is added to the partial plan, a constraint $(\langle x_1, \ldots, x_n, I_a \rangle, \gamma_{\text{act-inst}})$ is added to the binding constraint network, where I_a is the instantiation variable of a.

This instantiation variable has two main benefits. First it allows to find all possible instantiations of an action which is a useful input for the reasoning techniques we will introduce later in this section. This set of instantiations is iteratively refined through constraint propagation whenever the parameters of the action are updated. Second, when a ground action is known or found to be impossible, it can simply be removed from the allowed tuples in the relation. This change will trigger propagation in the binding constraint network and be reflected on the parameters of actions in the partial plan.

Similarly, every lifted task $\tau : \mathtt{tsk}(y_1, \ldots, y_n)$ is associated with a refinement variable R_τ that takes value in the ground actions whose task is tsk. At a given time, $dom(R_\tau)$ contains any ground action that might be used as a refinement for τ:

$$a \in dom(R_\tau) \iff task(a) = \texttt{tsk}(\texttt{c1}, \ldots, \texttt{cn}) \wedge \forall_{i \in 1..n}\ \texttt{ci} \in dom(y_i)$$

More specifically, a task symbol `tsk` is associated with a relation $\gamma_{\texttt{tsk-ref}}$ that gives the possible refinements of a task. If there is a possible ground action $id = \texttt{act}(\texttt{c1}, \ldots, \texttt{cn})$ whose task is $\texttt{tsk}(\texttt{p1}, \ldots, \texttt{pm})$, then the relation $\gamma_{\texttt{tsk-ref}}$ will contain the tuple $(\texttt{p1}, \ldots, \texttt{pm}, id)$. This identifies that the action (uniquely identified by id) is a possible refinement of the task $\texttt{tsk}(\texttt{p1}, \ldots, \texttt{pm})$. When a new unrefined task $\texttt{tsk}(y_1, \ldots, y_n)$ is inserted into the partial plan, it is associated with a refinement variable R_τ that must verify this relation, i.e., with a constraint $(\langle y_1, \ldots, y_n, R_\tau \rangle, \gamma_{\texttt{tsk-ref}})$.

When an action a is introduced as a refinement of task τ, their respective instantiation and refinement variables are unified by the introduction of a constraint $I_a = R_\tau$.

Instantiation and refinement variables make explicit the relationships between lifted actions and tasks in a partial plan and the ground actions that will appear in a solution plan. In search control, those variables are used to transform lifted partial plans into ground relaxed problem for reachability analysis and heuristic computation. The result of reachability analysis is also used to remove unreachable actions from the domains of both instantiation and refinement variables, as we will see in the next subsection.

3.3.2 Reachability Analysis

Reachability analysis has been a crucial component of many planning systems including FF [Hof01] and POPF [Col+10]. This analysis is done by solving a relaxed version of the planning problem in order to infer optimistic estimates of the set of states reachable from the initial state. It allows inferring which actions and fluents might appear in a solution plan. In classical planners, this analysis is often done by taking the delete-free relaxation of a planning problem. A set of reachable actions is then built by adding, one by one, any action whose preconditions are achieved either in the initial state or by an action in the reachable set.

This technique is not directly applicable to temporal problems that might contain inter-dependent actions (see Example 2.4). Indeed, if two actions A and B are inter-dependent (as in Figure 3.2), inferring that B is reachable requires knowing that A is reachable, which would require prior knowledge that B is reachable. Planners such as POPF go around this problem by further relaxing delete-free problems: each durative action is split in two instantaneous at-start and at-end actions. The at-start action does not contain the end conditions of the original action which eliminates any possibility of inter-dependency.

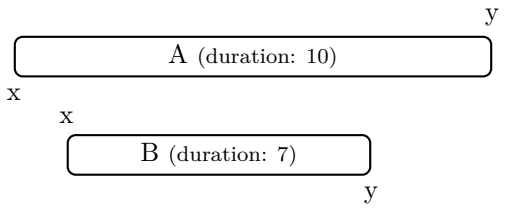


Figure 3.2: Two interdependent actions: A with a start effect x and an end condition y, and B with a start condition x and an end effect y.

Hierarchical problems introduce further complications to this scheme. For a satisfactory analysis, one should take into account the consequences of a hierarchized search space: that any high-level action should have its subtasks achieved and that any task-dependent action be required by a given task. Intuitively, these requirements of hierarchical problems lead to many interdependencies: a high-level action requires the presence of the actions achieving its subtasks and those actions can only appear in the plan if the high-level action initially required them (Figure 3.3).

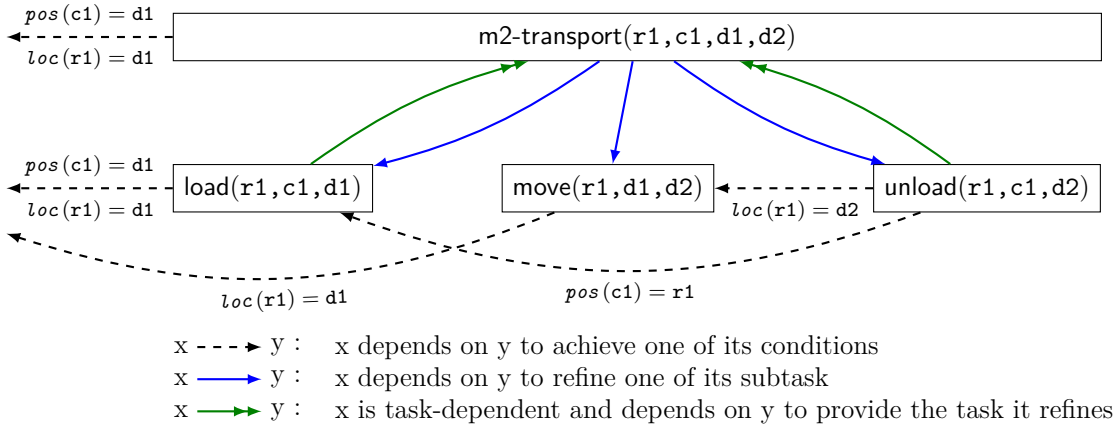


Figure 3.3: Dependencies between actions in a plan achieving a single `transport(c1,d2)` task. The m2-transport action is the one of Figure 2.2. Its three subtasks are refined by a *free* move action (Figure 2.1) and two *task-dependent* load and unload actions.

In this section, we describe a reachability analysis that takes into account the hierarchical properties of a problem and supports interdependent actions with no additional relaxation. More specifically, we consider delete-free actions as a basis for our relaxed model. To give an intuition of what a delete-free model would be, consider a change assertion $[t_s, t_e]\, sv : \mathtt{a} \mapsto \mathtt{b}$. This transition requires sv to have the value `a` at time t_s (i.e. that the fluent $\langle sv{=}\mathtt{a}\rangle$ holds at t_s) and states that sv will have the value `b` at time t_e, meaning that the $\langle sv{=}\mathtt{a}\rangle$ no longer holds at time t_e. This change assertion thus has a positive effect on the fluent $\langle sv{=}\mathtt{b}\rangle$ and a delete effect on the fluent $\langle sv{=}\mathtt{a}\rangle$. In a delete-free model, we would consider that after the change assertion the state variable has both the values `a` and `b`. Those delete-free actions are extended with additional conditions and effects that account for hierarchical features of the original actions.

This relaxed model is used to compute a set of actions that might appear in a solution together with information about their earliest appearance. This information is then used to infer constraints on a partial plans as well as to detect dead-ends in the search.

3.3.2.1 Relaxed Problem

In order to perform a reachability analysis, we start by defining a relaxed planning problem. A relaxed problem is a tuple $\langle F, A, I\rangle$ where F is a set of fluents, A is a set of elementary actions and I is the initial set of timed fluents.

The set of fluents F contains all fluents in the original domain (i.e. all combinations of a state variable and a value). Furthermore, for any task symbol τ in the original domain, F is extended with $\mathit{required}(\tau)$, $\mathit{started}(\tau)$ and $\mathit{ended}(\tau)$, which take a value in $\{\top, \bot\}$. Those respectively represent that an action achieving τ is needed, has started and has ended.

The elementary actions in A are simple temporal actions with a set of conditions and a single positive effect. Conditions and effects are on fluents in F and can represent causal or hierarchical requirements and effects of the original action.

The initial set of timed fluents I, represents a set of fluent whose appearance in a solution plan is already supported. It is built from both the set of *a priori* supported assertions in the original problem and from the effects of actions already in the partial plan.

Given a relaxed problem, the objective is to find the subsets of F and A that can be reached from the facts in I.

Elementary actions. We start by transforming our actions (both high-level and primitive) into a more traditional action representation with conditions and effects. Our objective is to obtain delete-free actions that still encompass temporal and hierarchical aspects of the original action. We associate any action a of the planning domain with a flat action a_{flat} together with a set of conditions $C_{a_{flat}}$ and effects $E_{a_{flat}}$ such that:

- for any persistence condition $\langle [t, t'] \, sv = v \rangle$ in a, a_{flat} has the condition $\langle [t] \, sv = v \rangle$
- for any assertion $\langle [t, t'] \, sv : v \mapsto v' \rangle$ in a, a_{flat} has the condition $\langle [t] \, sv = v \rangle$ and the effect $\langle [t'] \, sv = v' \rangle$
- for any a priori supported assertion $\langle [t] \, sv := v \rangle$ in a, a_{flat} has the effect $\langle [t] \, sv = v \rangle$
- given the task τ_a achieved by a, a_{flat} has the effect $\langle [t_{start}] \, started(\tau_a) = \top \rangle$ and the effect $\langle [t_{end}] \, ended(\tau_a) = \top \rangle$
- if a is task-dependent and its task is τ_a, then a_{flat} has the additional condition $\langle [t_{start}] \, required(\tau_a) = \top \rangle$
- for every subtask $\langle [t, t'] \, \tau \rangle$ of a:
  - two additional conditions $\langle [t] \, started(\tau) = \top \rangle$ and $\langle [t'] \, ended(\tau) = \top \rangle$
  - one additional effect $\langle [t] \, required(\tau) = \top \rangle$
- a_{flat} contains all constraints of a

Such flat actions define a relaxed version of the problem in which actions have no "delete effects" and hierarchical relationships are compiled as additional conditions and effects. It is important to note that the resulting 'flat' problem relaxes some hierarchical aspects of the original one. Indeed, a given subtask $\langle [st_\tau, et_\tau] \, \tau \rangle$ yields two conditions $[st_\tau] \, started(\tau) = \top$ and $[et_\tau] \, ended(\tau) = \top$. Those two conditions can be fulfilled by distinct actions, thus ignoring temporal constraints on the unique action that should have achieved the subtask in the original model. Furthermore, a single "subtask effect" could allow the presence of multiple task-dependent actions. This additional relaxation is however not a problem as this transformation is simply meant to expose hierarchical features of the problem for a reachability analysis that is optimistic.

The flat actions are still temporally complex and might feature numerous timepoints related by temporal constraints. As a second preprocessing step, those actions are split into simpler elementary actions where all delays are fixed. An elementary action contains a single effect and the necessary conditions to achieve this effect with a_{flat}. Specifically, given a flat action a_{flat}, the set of elementary actions for a_{flat} is given by:

$\mathsf{move}_{flat}(r, d, d')$

conditions:	$[t_{start}]\ \texttt{loc}(r) = d$
	$[t_{start}]\ \texttt{occupant}(d) = r$
	$[t']\ \texttt{occupant}(d') = \texttt{nil}$
effects:	$[t_{end}]\ \texttt{loc}(r) = d'$
	$[t]\ \texttt{occupant}(d) = \texttt{nil}$
	$[t_{end}]\ \texttt{occupant}(d') = \texttt{nil}$
	$[t_{start}]\ started(\mathtt{move}(r, d, d')) = \top$
	$[t_{end}]\ ended(\mathtt{move}(r, d, d')) = \top$
constraints:	$\texttt{connected}(d, d')$
	$t_{end} - t_{start} = 10$
	$t_{start} < t < t' < t_{end}$

Figure 3.4: Flattened version of the *move* operator of Figure 2.1. The implicit temporal constraints are shown explicitly here.

- for each effect $e = \langle [t_e] f \rangle$ in a_{flat}, creating a new elementary action a_e with $\langle [1] f \rangle$ as the only effect of a_e. In practice this places the effect exactly one time unit after the start of the elementary action. While not strictly needed, it facilitates reasoning of the relative placement of conditions and effect in an elementary action.

- any condition c in a_{flat} is added to each a_e with an optimistic timing constraint on when c is needed relatively to e. By optimistic, we mean the latest time at which c can be required to be true given the temporal constraints. For a condition $c = \langle [t_c] sv' = v' \rangle$ in a_{flat}, this is achieved by adding to a_e a condition $\langle [d] sv' = v' \rangle$ where d is the highest valid value that can be taken by t_c given that t_e took the value 1.

Figure 3.5 shows two elementary actions generated for the flat move of Figure 3.4. Three additional elementary actions are needed to cover the last three effects of the action.

Timed fluents. We now describe how the initial set of timed fluents I is derived from a partial plan. A initial timed fluent $i \in I$ is denoted by $\langle [t] f \rangle$ where f is a fluent and t is a time at which f holds. Given a partial plan $(\pi, \mathcal{F}, \mathcal{C})$, I is composed of:

- for any assertion $\langle [t_1, t_2]\, sv : v_1 \mapsto v_2 \rangle$ or $\langle [t_2]\, sv := v_2 \rangle$ in $\mathcal{F}$; if sv' and v_2' are instantiations of sv and v_2 consistent with $\mathcal{C}$, I contains $\langle [t_2']\, sv' = v_2' \rangle$ where t_2' is the smallest instantiation of t_2 consistent with $\mathcal{C}$, given by $dist_{STN}(\mathcal{O}, t_2)$.

- for any unrefined task $\langle [t_1, t_2]\, \tau \rangle$ in π; if τ' is an instantiation of τ consistent with $\mathcal{C}$, I contains $\langle [t_1']\, required(\tau') = \top \rangle$ where t_1' is the smallest possible instantiation of t_1 consistent with $\mathcal{C}$.

Definitions. An elementary action is *applicable* once all its conditions are met. An action with an effect f is called an *achiever* of the fluent f. A fluent becomes *achievable* after one of its achievers becomes applicable. As a consequence of using (delete-free)

$\mathsf{move}_{loc(r)}(r, d, d')$
conditions: $[-9]$ $\texttt{loc}(r) = d$
$[-9]$ $\texttt{occupant}(d) = r$
$[0]$ $\texttt{occupant}(d') = \texttt{nil}$
$[-9]$ $required(\mathtt{move}(r, d, d')) = \top$
effects: $[1]$ $\texttt{loc}(r) = d'$
constraints: $\texttt{connected}(d, d')$

$\mathsf{move}_{occupant(d)}(r, d, d')$
conditions: $[0]$ $\texttt{loc}(r) = d$
$[0]$ $\texttt{occupant}(d) = r$
$[9]$ $\texttt{occupant}(d') = \texttt{nil}$
$[0]$ $required(\mathtt{move}(r, d, d')) = \top$
effects: $[1]\texttt{occupant}(d) = \texttt{nil}$
constraints: $\texttt{connected}(d, d')$

Figure 3.5: The first two elementary actions generated from $move_{flat}$

elementary actions, once a fluent is achievable or an action is applicable, it stays achievable/applicable at all subsequent time points.

Action a is applicable at time t (denoted by $applicable(a, t)$) if for all conditions $\langle[\delta]f\rangle$ of a, f is achievable at time $t + \delta$. Similarly, a fact f' is achievable at time t' (noted $achievable(f', t')$) if there exists an achiever of f' applicable at time $t' - 1$.

We say that an action a (resp. a fluent f) is reachable if there exists a time t such that $applicable(a, t)$ holds (resp. $achievable(f, t)$ holds). The *earliest appearance* of a reachable action a (denoted by $ea(a)$) is the smallest t for which $applicable(a, t)$ holds. Similarly, the earliest appearance of a reachable fluent f is the lowest t for which $achievable(f, t)$ holds.

3.3.2.2 Reachability analysis with inter-dependent actions

The problem of inter-dependent actions. As shown by Cooper, Maris, and Régnier [CMR13], the difficulty of doing reachability analysis with interdependent actions is due to the presence of *after-conditions*: conditions that must hold after an effect of the action is achieved. For instance, the end condition y of the action A of Figure 3.2 is an *after-condition*. In our elementary actions, an *after-condition* can be easily detected as any condition $\langle[t]\, f\rangle$ where $t \geq 1$. All other conditions are referred to as *before-conditions*.

The approach taken by POPF of splitting an action into instantaneous at-start and at-end actions in practice means that all after-conditions are ignored. In Figure 3.2, the action A would indeed become action A_{start} containing only an effect x and an action A_{end} with a condition y. In this model, the effect x can thus be produced independently of the after-condition on y. This constitutes an additional relaxation resulting in the elimination of all interdependencies in the delete-free problem.

While this relaxation is reasonable for many generative planning problems, hierarchical problems typically feature many interdependencies between methods and their subtasks. The remaining of this section describes a technique for reachability analysis that does not need any additional relaxation, thus taking all after-conditions into account.

Algorithm 2 Algorithm for identifying reachable actions and fluents and computing their earliest appearance.

$A \leftarrow$ Elementary actions
$F \leftarrow$ Fluents
$I \leftarrow$ Initial timed fluents
$Q \leftarrow \varnothing$ ▷ Priority queue of $\langle action/fluent, time\rangle$ ordered by increasing $time$
for all $f \in F$ **do**
    $reachable(f) \leftarrow \bot$
for all $\langle [t]\, f\rangle \in I$ **do**
    $Q \leftarrow Q \cup \{\langle f, t\rangle\}$
for all $a \in A$ **do**
    $reachable(a) \leftarrow \bot$
    if a has no before-conditions **then**
        $Q \leftarrow Q \cup \{\langle a, 0\rangle\}$

while Q non empty **do**
    DijkstraPass
    for all $a \in$ A **do**
        for all $\langle [\delta]\, f\rangle \in$ after-conditions of a **do**
            if $\neg reachable(f)$ **then**
                $\langle A, F\rangle \leftarrow$ RecursivelyRemove(a)
            else if $ea(a) < ea(f) + \delta$ **then**
                $Q \leftarrow Q \cup \{\langle a, ea(f) + \delta\rangle\}$
    for $a \in A$ **do**
        if a is late **then**
            $\langle A, F\rangle \leftarrow$ RecursivelyRemove(a)

procedure DijkstraPass
    while Q non empty **do**
        $\langle n, t\rangle \leftarrow pop(Q)$
        if n already expanded in this pass **then**
            continue
        if $reachable(n) \wedge ea(n) \geq t$ **then**
            continue
        $reachable(n) \leftarrow \top$
        $ea(n) \leftarrow t$
        if n is an action with the effect $\langle [\delta]\, f\rangle$ **then**
            $Q \leftarrow Q \cup \{\langle f, t + \delta\rangle\}$
        else
            for all $a \in A$ with a condition on the fluent n **do**
                if all before conditions of a are reachable **then**
                    $t' \leftarrow max_{\langle [\delta]\, f\rangle \in C_a}\ ea(f) - \delta$
                    $Q \leftarrow Q \cup \{\langle a, t'\rangle\}$

Propagation To handle after-conditions during reachability analysis, as detailed in Algorithm 2, we alternate two steps: *(i)* a propagation that ignores all after-conditions by performing a Dijkstra like propagation in the graph composed of all fluents and elementary actions; and *(ii)* a second step that enforces all after-conditions. Those two steps are complemented with a pruning mechanism that repeatedly detects actions that have unsolvable interdependencies.

Algorithm 2 begins by selecting a set of *assumed reachable* elements (i.e. actions or fluents) from which to start propagation (lines 4-12). The obvious candidates are fluents known to be true at a given time, e.g., fluents achieved by assertions in the problem definition or by actions in the partial plan. All such fluents have been previously inserted the initial set of timed fluents (I) and are selected. We also optimistically select all actions that have no *before-conditions*, i.e., actions whose all conditions are after-conditions. Those assumed reachable elements are inserted into a priority queue Q of $\langle n, t\rangle$ pairs where n is either an elementary action or a fluent and t is a candidate time for its earliest appearance.

The initial assumed reachable set is then iteratively extended with all fluents with an assumed reachable achiever and any action whose all before-conditions are assumed reachable. This is done by a Dijkstra-like propagation (line 15), that extracts the items in Q by increasing earliest appearances. The corresponding actions (resp. fluents) are marked as reachable and the fluents (resp. actions) depending on them are inserted into Q. More specifically, if a pair $\langle a, t\rangle$ is extracted from Q and a is an action with the effect $\langle [\delta]\, f\rangle$, the pair $\langle f, t+\delta\rangle$ is inserted into the queue. If a pair $\langle f, t\rangle$, with $f \in F$, is extracted from Q, all actions depending on f that have all their before-conditions reachable are pushed in Q (lines 38-41).

As a second step, we revise our optimistic assumptions by considering after-conditions:

- Line 19 removes any action a with an after-condition on an unreachable fluent f. More specifically, the RecursivelyRemove procedure marks its parameter as unreachable and removes it from the set of actions. The removal is recursive: if a removed action is the only achiever for a fluent f then f is removed as well (and as a consequence all actions depending on f will also be removed, etc.). Furthermore, if the first achiever of a fluent is removed from the graph and there is at least one other achiever for it, then the fluent is added back to Q with an updated earliest appearance.

- Line 21 takes an after-condition of an action a on a reachable fluent f and enforces the minimal delay δ between $ea(f)$ and $ea(a)$. If the current delay is not sufficient, a is added to Q and will be reconsidered upon the next Dijkstra pass.

Finally, *late actions* are marked unreachable and removed from the graph (line 24). We say that an action a is *late* if for any *non-late* action a', $ea(a') + d_{max} < ea(a)$ where d_{max} is the highest delay in the relaxed model (either of a timed initial literal or between a before-condition and an effect). In practice, this means that actions are partitioned into non-late and late, these two sets being separated by a temporal gap of at least d_{max}. The intuition (demonstrated in the next subsection) is that the earliest appearance of a late action is being pushed back due to unachievable interdependencies with other late actions.

The two-step process is repeated (line 14) to take into account the newly updated reachability information. In the subsequent runs, Dijkstra algorithm will start propagating from the items updated by the previous iteration, with lines 31-32 making sure that

the earliest appearance values $ea(n)$ are never decreased to a too optimistic value. The algorithm detects a fix-point and exits if the queue is empty, meaning that after-conditions did not trigger any change.

3.3.2.3 Analysis and Possible Variants

We now explore some of the characteristics of Algorithm 2. The first Dijkstra pass acts as an optimistic initialization: it identifies a set of possibly reachable nodes and assigns them earliest appearance times. All operations after this first pass will only *(i)* shrink the set of reachable nodes; and *(ii)* increase the earliest appearance times.

For analysis, it is helpful to see the relaxed problem as a graph whose nodes are the fluents and elementary actions. Edges either represent a condition (edges from a fluent to an action) or an effect (edges from an action to a fluent).

Definition 3.3.1 (Causal loop)**.** We denote as a causal loop a cycle of actions and fluents $f_0 \to A_0 \to f_1 \to A_1 \dots A_n \to f_0$, such that each fluent f_i is a condition of the elementary action A_i and each action A_i is an achiever of the fluent f_{i+1}.

Each edge of this loop is associated with a delay d that is respectively the delay from when an action A_i start to the moment its effect f_{i+1} is achieved, or the delay from when a condition f_i is needed to the moment its containing action A_i can start.

The notion of causal loop is crucial in the understanding of problems with interdependent actions. We say that a causal loop is *self-supporting* if the sum of the delays on its edges is less than or equal to 0. The actions of Figure 3.2 form a self-supporting causal loop $A \xrightarrow{0} x \xrightarrow{0} B \xrightarrow{7} y \xrightarrow{-10} A$, which essentially means that B can be used to produce the condition y of A early enough for A to be executable.

A causal loop is said to be *unfeasible* if its length is strictly positive. If B had a duration of 12 in Figure 3.2, we would have an unfeasible causal loop $A \xrightarrow{0} x \xrightarrow{0} B \xrightarrow{12} y \xrightarrow{-10} A$. Indeed, B does not "fit" in A anymore and the planner must find another way to achieve either x or y to use those two actions.

Proposition 3.3.1. *If a node (i.e. action or fluent) n is reachable in the relaxed problem, then $ea(n)$ converges to a finite value. If a node n' is not reachable then $ea(n')$ either remains at ∞ or diverges towards ∞ until it is removed from the graph.*

Proof (Sketch). We sketch the proof that is fully given in Appendix A.1. An action or fluent n is reachable if there is either a path from initial facts to n or if n is part of a self-supporting causal loop (i.e. cycle of negative or zero length). Consequently and because the earliest appearance can only increase, repeated propagations will eventually converge. On the other hand, an unreachable node either depends on an unreachable node or is involved only in causal cycles of strictly positive length. If the node was ever assumed reachable, its earliest appearance will thus be increased by Algorithm 2 until it is removed from the graph. □

Proposition 3.3.2. *If a node is put in the late set, then it is not reachable.*

Proof (Sketch). We sketch the proof that is fully given in Appendix A.2. The intuition is that the gap between non-late and late nodes appeared because late nodes are delaying each other due to positive causal cycles. We first show that any late node was delayed to its current time due to a dependency on another late node: because the temporal gap

is bigger than all delays in the model, a non-late node could not have influenced a late node. It follows that any late node depends on at least one other late node. Furthermore a late node necessarily participates in a positive cycle or depends on a late node that does. From there, one can show that at least one node n in this group is involved only in positive cycles. Any other possibility (path from timed initial literals or negative cycle) would have resulted in n being less than d_{max} away from a non-late node. □

It follows from propositions 3.3.1 and 3.3.2 that Algorithm 2 produces a reachability model (denoted as R_∞) that contains a fluent or action n and its earliest appearance $ea^*(n)$ iff n is reachable in the relaxed problem. In the worst case, computing this model has a pseudo-polynomial complexity since there may be as many as d_{max} iterations of the algorithm (d_{max} being the highest delay in the relaxed model). The worst-case complexity of each iteration is dominated by the Dijkstra pass of $O(|N| \times \log(|N|) + |E|)$, where N is the number of fluents and actions and E is the number of conditions and effects appearing in actions.

Discussion. One might consider computing various approximations of R_∞ by limiting the number of iterations to a fixed number K, making the algorithm polynomial in $O(K \times |N| \times \log(|N|) + |E|)$ for producing a reachability model R_K. In the special case where $K = 1$, this is equivalent to performing a single Dijkstra pass and removing all actions with an unreachable after-condition. Increasing K would allow us to better estimate the earliest appearances and detect additional late nodes.

Another simplification is to ignore all after-conditions, which can be done by stopping Algorithm 1 after the first Dijkstra pass. In practice, this model has all the characteristics of the relaxed temporal planning graph of POPF: (1) the separation of durative actions into at-start and at-end instantaneous actions is done by the transformation into elementary actions; (2) the minimal delay between matching at-start and at-end actions is enforced by the presence of start conditions in the elementary actions representing the end effects; and (3) any end condition appearing in the elementary action of a start effect would be ignored because it would be an after-condition. We refer to this reachability model as R^+.

It is interesting to note that R^+ and R_∞ are equivalent on all problems with no after-conditions. Classical planning obviously falls in this category as well as any PDDL model with no at-start effect or no at-end condition. In fact, on such problems R^+ and R_∞ are equivalent to building a relaxed temporal planning graph, with no significant computational overhead.

3.3.2.4 Using the results of a reachability analysis

For a given partial plan ϕ, a reachability analysis provides us with:

- Ra_ϕ, a set of actions reachable in the relaxed problem,
- Rf_ϕ, a set of fluents reachable in the relaxed problem,
- $ea_\phi : (Ra_\phi \cup Rf_\phi) \to \mathcal{N}$ a function associating each reachable action and fluent with an optimistic earliest time at which it can be added or achieved in a solution plan.

Those are computed for any partial plan that is extracted from the priority queue for expansion. Because all computed values are optimistic, Algorithm 2 can be run incrementally by initializing the set of reachable nodes and earliest appearances with those computed for the previous partial plan. While the complexity of the incremental version is unchanged, our implementation suggests that it avoids a lot of redundant computations. The results of a reachability analysis are used in many parts of the planner to prune parts of the search space and derive additional constraints on the current partial plan:

- For any unsupported assertion $\alpha \in \mathcal{F}_\phi$, if its condition cannot be instantiated to a reachable fluent $f \in Rf_\phi$, then the partial plan is marked as a dead-end and search proceeds with the next best partial plan. Otherwise we temporally constrain α to be at least as late as its earliest reachable instantiation. This is done by adding the following constraint to the STN:

$$dist_{STN}(\mathcal{O}, start(\alpha)) \geq \min\{\, ea(f) \mid f \in Rf_\phi \cap dom(cond(\alpha)) \,\}$$

- We check that all unrefined tasks can be refined by a reachable action. This is done by restricting the domain of any refinement variable to reachable actions:

$$dom(R_\tau) \subseteq Ra_\phi$$

  If a task has no possible refinement (i.e. one refinement variable has an empty domain) then the partial plan is declared a dead-end. Otherwise, the earliest start time of all unrefined tasks is updated to be at least as late as the earliest reachable action that can refine it.

- When considering unsupported assertions or unrefined tasks flaws, we disregard any resolver involving an action with no reachable instances. For instance, if there is no instance of the move action in Ra_ϕ, then the planner would not consider the insertion of move to support an assertion on the location of a robot. In this case, the planner would need to rely on assertions already in the partial plan.

- All domain transition graphs (that will be introduced in Section 3.3.3) are updated by removing any transition provided by an unreachable action. This update has indirect effects, since it allows more reliable information when reasoning on causal networks.

- When creating the instantiation variables of Section 3.3.1, the domain of such variables is constrained to be a subset of Ra_ϕ. This indirectly constrains the parameters of any newly added action to respect some reachability requirements.

3.3.3 Causal Network

We define the causal network of a partial plan ϕ as the graph $G_\phi = \langle N, E \rangle$ where N is the set of assertions in $\mathcal{F}_\phi$ and E contains an edge $x \to y$ iff there is a causal link stating that x supports y. This causal network is explicitly maintained by the planner by adding edges when new causal links are inserted and adding nodes when new assertions are introduced by newly added actions. For a partial plan to be a solution, the corresponding causal network must be such that:

- every assertion $x \in N$ that is not a priori supported has an incoming edge
- any change assertion or a priori supported assertion $x \in N$ has at most one outgoing edge that targets a change assertion. In addition, x might support several persistence conditions.

In this section, we show how this graph G_ϕ can be exploited to infer additional constraints on the partial plan and extract heuristic information.

Definition 3.3.2 (Causal Chain)**.** A causal chain is a sequence of change assertions $\langle \beta_1, \ldots, \beta_n \rangle$ such that for any element β_i there is a causal link to its direct successor β_{i+1}.

A causal chain spans over the temporal interval $[start(\beta_1), end(\beta_n)]$ and is said to be about the state variable sv common to all its composing assertions.

We say that two causal chains *possibly overlap* if their state variables can be unified by consistent binding constraints and they span over two possibly overlapping temporal intervals. Two causal chains *necessarily overlap* if every consistent instantiations overlap. Two necessarily overlapping causal chains result in an unsolvable threat because at least one change assertion of the first chain would temporally overlap a change assertion or a causal link of the second chain.

In order to facilitate the reasoning on the possible transitions that can be taken by a state variable, we now introduce Domain Transition Graphs.

Definition 3.3.3 (DTG)**.** A Domain Transition Graph (DTG) of a state variable sv is a directed graph (N, E) where N is composed of the values that can be taken by sv and a special node ANY. E is a set of allowed transitions from one value to another. An edge in E is of the form $v_1 \xrightarrow{d} v_2$ meaning that the value of sv can be changed from v_1 to v_2 in d time units. In the special case where $v_1 = \text{ANY}$, it means that sv can take the value v_2 regardless of its previous value (even if sv had no known previous value).

The DTG of a given state variable sv is built as follows. For any ground action a that is reachable (according to reachability analysis):

- if the action contains a change assertion $\langle [t_1, t_2]\, sv : v_1 \mapsto v_2 \rangle$, then the DTG contains an edge $v_1 \xrightarrow{t_2 - t_1} v_2$
- if the action contains an a priori supported assertion $\langle [t]\, sv := v \rangle$, then the DTG contains the edge $\text{ANY} \xrightarrow{1} v$.

We say that there is a *feasible transition* of a state variable sv from a value v_1 to a value v_2 if there is a path in the DTG of sv from v_1 to v_2 or from ANY to v_2. We denote as $dist_{DTG}(v_1, v_2)$ the length of the shortest such path. An example of a DTG is given in Figure 3.6.

3.3.3.1 Potential supporters

In order to appear in a solution plan, any unsupported assertion must eventually be linked to a supporting assertion. This link can take the form of a single causal link or of a chain of causal links going through statements not yet in the plan. We refer to the candidates for such supporting assertions as *potential supporters*.

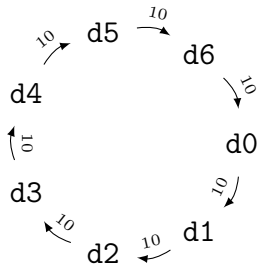


Figure 3.6: Example DTG of $\texttt{loc}(r)$: where the location in which the robot r can navigate are organized in a circular pattern. Moving from one place to the next takes 10 time units.

Definition 3.3.4 (Potential supporter)**.** Given a causal network G_ϕ of a partial plan ϕ, a change assertion β is a *potential supporter* of an unsupported assertion α if a set of statements $\{s_1, \ldots, s_n\}$ and a chain of causal links $\beta \to s_1 \to \ldots \to s_n \to \alpha$ can be added to G_ϕ.

For an unsupported assertion α, we consider a superset of the set of potential supporters, noted $\mathcal{S}_\alpha$. This set is incrementally updated to contain any change assertion $\beta \in N$ that satisfies the following necessary conditions:

- the state variables of α and β are unifiable.
- there is a feasible transition from the value produced by β to the value required by α.
- adding a chain of causal link from β to α will not result in any unsolvable threat. We consider that there is an unsolvable threat, if the causal chain obtained by concatenating the current causal chains of α and β would necessarily overlap an existing causal chain.

In the causal network example of Figure 3.7, the potential supporters of α would be β and μ and the potential supporters of γ would be β and α. The potential supporters of ρ are α and ρ. Indeed any causal chain from β to ρ would be threatened by α and by the causal chain of γ and μ.

3.3.3.2 Deriving Constraints from Possible Supporters

We now consider what temporal constraints can be inferred from the necessary evolution of a causal network. In order to keep the explanations and notations concise, we first assume that actions do not contain any *a priori* supported assertions.

Given this assumption, an unsupported assertion α must eventually be linked to a change assertion $\beta \in \mathcal{S}_\alpha$ by a chain of causal links $\beta \to \cdots \to \alpha$. The length of the causal chain depends on the change assertions needed to go from the value produced by β (noted $\mathit{eff}(\beta)$) to the value needed by α (noted $\mathit{cond}(\alpha)$). Therefore, $\mathit{start}(\alpha)$ must be after $\mathit{end}(\beta)$ with a delay depending on the values of $\mathit{eff}(\beta)$ and $\mathit{cond}(\alpha)$. More formally, this requirement is expressed by the following inequality,

$$dist_{STN}(\mathcal{O}, start(\alpha)) \geq \min_{\beta \in \mathcal{S}_\alpha} dist_{STN}(\mathcal{O}, end(\beta)) + dist_{DTG}(\mathit{eff}(\beta), cond(\alpha))$$

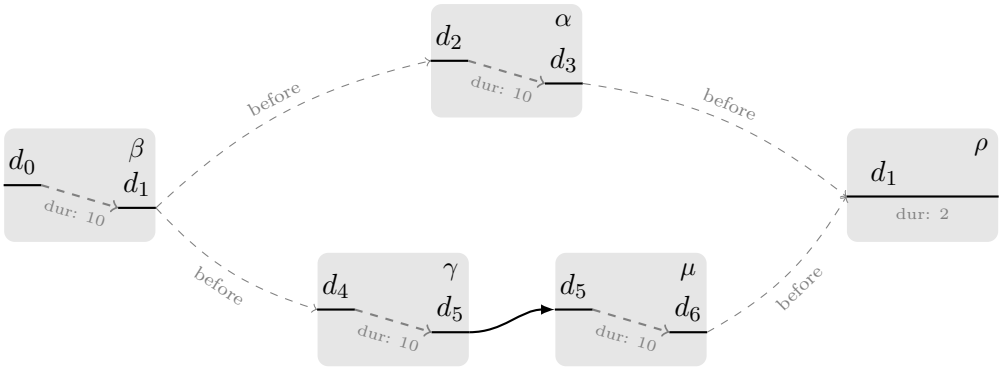


Figure 3.7: Partial view of a causal network of a state variable $\texttt{loc}(r)$ with 4 change assertions $(\beta, \alpha, \gamma, \mu)$ and one persistence conditions ρ, all on the same state variable $loc(r)$. There is a causal link from γ to μ, β is temporally constrained to be before α and γ, ρ is temporally constrained to be after α and μ. We further suppose β to be initially supported. This causal network is to be considered jointly with the DTG of Figure 3.6.

where $dist_{STN}(\mathcal{O}, t)$ is the minimal delay in the STN between the origin of time $\mathcal{O}$ and the time point t and $dist_{DTG}(x, y)$ represents the length of the minimal path in the DTG to go from any instantiation of x to any instantiation of y. If this inequality does not hold, it is enforced by setting the earliest time of $start(\alpha)$ to be greater or equal to the right side of the inequality.

In the case where an assertion α has a single potential supporter β, one can devise a more specific version that does not use a triangular inequality:

$$dist_{STN}(end(\beta), start(\alpha)) \geq dist_{DTG}(\mathit{eff}(\beta), cond(\alpha))$$

Again this inequality is enforced by adding in the STN a minimal delay constraint between et_β and st_α.

In the case where some actions contain an *a priori* supported assertion and that α can be achieved using one such assertion (i.e. $dist_{DTG}(\text{ANY}, cond(\alpha)) \neq \infty$), the above rules are generalized by considering a virtual *possible supporter* that could support it at time $dist_{DTG}(\text{ANY}, cond(\alpha))$.

Example 3.3. Let us now consider what applying those rules on the causal network of Figure 3.7 would allow us to infer. Assuming that $dist_{STN}(\mathcal{O}, st_\beta) = 0$, we would infer the following temporal constraints:

$$\begin{aligned}
dist_{STN}(\mathcal{O}, st_\alpha) &\geq \min\ \{\, dist_{STN}(\mathcal{O}, et_\beta) + 10,\ dist_{STN}(\mathcal{O}, et_\mu) + 30 \,\} \\
&\geq 20 \\
dist_{STN}(\mathcal{O}, st_\gamma) &\geq \min\ \{\, dist_{STN}(\mathcal{O}, et_\beta) + 30,\ dist_{STN}(\mathcal{O}, et_\alpha) + 10 \,\} \\
&\geq 40 \\
dist_{STN}(\mathcal{O}, st_\rho) &\geq \min\ \{\, dist_{STN}(\mathcal{O}, et_\alpha) + 50,\ dist_{STN}(\mathcal{O}, et_\mu) + 20 \,\} \\
&\geq 80
\end{aligned}$$

The important catch is the detection that the persistence condition $\rho = \langle loc(r) = d_1 \rangle$ cannot be satisfied before time 80. Indeed, the planner has

already made commitments to other change assertions, between when the value d_1 is first achieved by β and the moment it is required by ρ.

3.3.3.3 Estimating the number of additional assertions needed for a valid causal chain

We further use the causal network as part of heuristic evaluation in order to estimate how many additional assertions are needed to support the set of open goals.

For each open goal α, the key idea is to find a chain of causal links going from an *a priori* supported assertion to α. We seek a minimal chain: filling out the missing parts should result in as few additional assertions as possible. Figure 3.8 gives an example of the minimal causal chain needed to support the persistence condition ρ from Figure 3.7. Building such a causal chain requires the addition of 4 change assertions, resulting in as many new open goals to be solved.

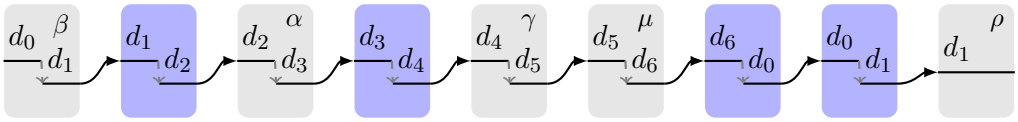


Figure 3.8: One possible causal chain to support the persistence condition $\rho = \langle loc(r) = d1 \rangle$. In blue (and with no label in the upper right) are temporal assertions that would need to be added for the causal chain to be complete.

Let us now define how we compute $hc(\alpha)$, an estimation of the number of additional assertions that are needed to build a complete causal chain to the open goal α. We first remark that our lifted representation means that there are multiple candidates for the instantiation of the condition of α. We thus introduce $hc(f, \alpha)$ as the cost of building the causal chain to α if its condition is the ground fluent f. Since the planner has the choice in the instantiation of variables, we consider the cost of building the causal chain to α to be the minimum of the cost of building it with any possible instantiation of its condition:

$$hc(\alpha) = \min_{\langle sv=v \rangle \in dom(cond(\alpha))} hc(\langle sv=v \rangle, \alpha) \tag{3.1}$$

where $hc(f, \alpha)$ is the cost of achieving α if its condition is f and $dom(cond(\alpha))$ is the set of possible instantiations of the condition of α.

Let us now define $hc(f, \alpha)$, the cost of achieving the ground condition $f = \langle sv=v \rangle$ of an assertion α. We give a recursive definition: the cost of building a causal chain is expressed as the cost of having its rightmost link plus the cost of building a chain up to this rightmost link.

$$hc(\langle sv=v \rangle, \alpha) = \begin{cases} 0 & \text{if } \alpha \text{ initially supported} \\ \min\limits_{\langle sv=v' \mapsto v) \rangle \in dom(\beta)} hc(\langle sv=v' \rangle, \beta) & \text{if } \exists \text{ causal link } \beta \to \alpha \\ \min \begin{cases} \min\limits_{\gamma \in \mathcal{S}_\alpha,\ \langle sv=v' \mapsto v \rangle \in dom(\gamma)} hc(\langle sv=v' \rangle, \gamma) \\ \min\limits_{e=\langle v' \to v \rangle \in DTG(sv)} c(e) + hc(\langle sv=v' \rangle, \alpha) \end{cases} & \text{otherwise} \end{cases} \tag{3.2}$$

Intuitively, there is no additional cost if α is *a priori* supported because there is no need for any causal support (i.e. $hc(\cdot, \alpha) = 0$).

If α is supported by an incoming causal link $\beta \to \alpha$, this causal link is necessarily the last link of the causal chain to α (for instance in Figure 3.7, the last link of any causal chain to μ is the existing causal link $\gamma \to \mu$). Thus, the cost of achieving the condition f of α is the cost of achieving the condition f' of β, where f' is an instantiation of the condition of β such that β produces f.

If α is in neither of these cases, we are left with two possibilities for the last link of its causal chain. First, if α has a possible supporter $\gamma \in \mathcal{S}_\alpha$ and γ can be instantiated to provide f, then a possibility is to have a causal link $\gamma \to \alpha$. In this case, the cost is the one of achieving γ with such an instantiation. Second, there might a possible action achieving $f = \langle sv{=}v \rangle$ by adding a change assertion of the state variable sv from a value v' to v. Such a change would appear as an edge e in the DTG of sv. As adding this link in the causal chain requires inserting a new action in the plan, we associate a cost $c(e)$ to this operation. This cost is set to the number of unsupported assertions in the introduced action, e.g., for the action move of Figure 2.1 this cost would be 3 since its insertion would result in 3 new assertions. Furthermore, one still needs to build the causal chain as to achieve the value $f' = \langle sv{=}v' \rangle$ for α.

Example 3.4. Considering the causal network of Figure 3.7, the equation below gives the estimated cost of building a causal chain to α (i.e. $hc(\alpha)$). Since all variables in α are already bound, there is a single possibility for instantiating its condition (first line). From there, the only possibility to provide the value d2 is to insert an additional $\mathsf{move}(r, \mathtt{d1}, \mathtt{d2})$ action, resulting in an additional cost of 3 (second line). To provide the value d1, we can either have a causal link from β or add another action $\mathsf{move}(r, \mathtt{d0}, \mathtt{d1})$ again with an additional cost of 3 (third line). Since β is initially supported, it induces no extra cost and we can conclude that $hc(\alpha) = 3$: building a complete causal chain to support α would require the insertion of three new assertion into the partial plan.

$$\begin{aligned} hc(\alpha) &= hc(\langle loc(r){=}\mathtt{d2} \rangle, \alpha) \\ &= 3 + hc(\langle loc(r){=}\mathtt{d1} \rangle, \alpha) \\ &= 3 + \min \{\ hc(\langle loc(r){=}\mathtt{d0} \rangle, \beta),\ \ 3 + hc(\langle loc(r){=}\mathtt{d0} \rangle, \alpha)\ \} \\ &= 3 + \min \{\ 0, \dots\ \} \\ &= 3 \end{aligned}$$

For computing $hc(\alpha)$, we use a distance computation in an equivalent graph where each node is a pair $\langle f, \alpha \rangle$, f being a fluent and α an assertion in the causal network. An example of such a graph for the causal network of Figure 3.7 is given in Figure 3.9. Edges in the graph represent the different possible transitions defined in equation (3.2). We distinguish a causal link from an existing assertion (two-headed green with a cost of 0) and a causal link from an additional assertion (red with a non-zero cost). We consider two special kinds of nodes: nodes representing *a priori* supported conditions (empty circles) and nodes representing unsupported conditions (blue circles). Finding a minimal causal chain to an unsupported assertion α is equivalent to finding in the graph the shortest path from any *a priori* supported node to any node representing an instantiation of α.

In the example of Figure 3.9, we can easily find the causal chain of Figure 3.8 by

looking for the shortest path from $\langle d_0, \beta \rangle$ to $\langle d1, \rho \rangle$. This shortest path takes 4 red edges for a final cost of 12, allowing us to conclude that $hc(\rho) = 12$.

In practice, the computation of $hc(\alpha)$ is done by a backward Dijkstra search: initializing the priority queue with nodes $\{\langle f, \alpha \rangle | f \in dom(cond(\alpha))\}$. Search proceeds by selecting the node with the least cost in the priority queue and adding its direct ancestors to the queue with an updated cost. Search continues until a node $\langle f', \beta \rangle$, where β is *a priori* supported, is extracted from the queue. The cost of this node gives the cost of the minimal causal chain to α (i.e. $hc(\alpha)$).

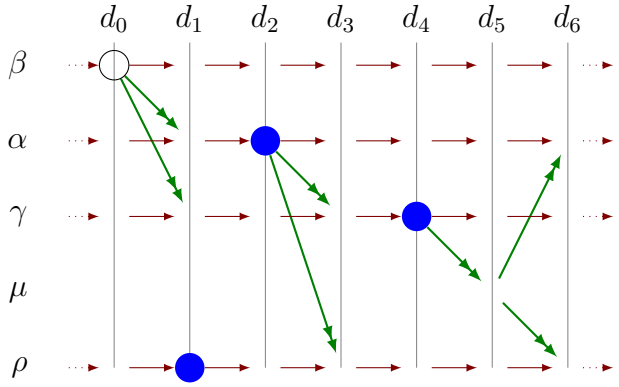


Figure 3.9: Virtual graph used for computing the minimal causal chain of the assertions of Figure 3.7.

3.3.4 Search Strategies

The search algorithm is responsible for choosing which node of the search tree to expand in order to quickly find a solution of good quality. The quality of the solution and the time spent finding it are often conflicting objective. By default in FAPE, the priority is given to the latter and plan quality is left as a secondary objective, used as a tie breaking criteria. The general motivation behind this choice is that the quality of a plan is often a domain-dependent criteria that can for instance include: the number of actions in the plan, the makespan of the plan, the work load distribution between the different agents or the presence of certain action sequences in the plan. For this reason, we deliberately focus on finding a valid plan independently of its cost. Of course, FAPE provides the capability to consider cost as an additional objective of search either through anytime planning or by defining domain-dependent heuristic but we do not discuss those possibilities here.

As it is usually the case in plan-space planning, our search procedure (Algorithm 1) requires two choices to be made at each search iteration. The first one is the nondeterministic choice of which partial plan to consider next that will define the order in which the search space is explored. The second one is the choice of a flaw to be fixed in the selected partial plan. As all flaws must eventually be fixed for a partial plan to become a solution, this choice is not a backtracking point but will have an impact on the shape of the search space. A search strategy is composed of two schemes that dictate the choices made in those two cases.

As it can be expected, a good strategy is not universal as it must take into account many specificities of the problem at hand. Because we support such a wide range of planning problem, from generative to HTN problem, we define two strategies. The first

one aims at being very general and have good performance on a wide range of problem while the second is specifically tailored for fully hierarchical problems.

3.3.4.1 General Search Strategy

Plan selection. Our plan selection strategy is conceptually based on $A\epsilon$ [GA83] in that it contains two queues both sorted by the same priority function f. The first one Q_{All} contains all partial plans that have been generated and not expanded. The second one Q_{Chi} is a subset of Q_{All} that is limited to children of the last expanded node. Those two queues serve different purposes that can be seen as diversification versus intensification: the planner chooses the partial plan to consider next either as the globally most promising according to f or commits to its previous choice and tries to further advance the last chosen plan.

The choice of the queue to use is governed by a parameter ϵ and is done as follows: if $\min_{\phi \in Q_{All}} f(\phi) < (1+\epsilon) \times \min_{\phi \in Q_{Chi}} f(\phi)$ then the next partial plan is the one with the lowest f value in Q_{All}. Otherwise it is the one with the lowest f value in Q_{Chi}. In practice, it means the search is restricted to the children of the last explored nodes until they are significantly worse than the globally best partial plan, where the threshold of "significantly" is given by ϵ. The objective of this technique is to compensate the inaccuracies in the definition of f.

The priority function $f(\phi)$ is defined as the sum of the following values:

1. the number of assertions in $\mathcal{F}_\phi$. This helps in estimating the search effort already done. It can been seen as normalized version of the number of actions in the plan as other parts of f use the number assertions rather than the number of actions.

2. the number of unrefined task in π_ϕ. This conservatively estimates the search effort left due to unrefined tasks.

3. the number of unsupported assertions in $\mathcal{F}_\phi$.

4. the number of assertions involved in at least one threat. This is a conservative estimation of the search effort left due to threats. We do not consider the number of threats itself as it can be, in the worst case, quadratic in the number of assertions in the plan, resulting in a very important impact on the value of f. Furthermore the addition of a single temporal constraint is likely to solve many threats at once.

5. the expected number of assertions that must be added to the partial plan. This is computed as the sum, for every unsupported assertion $\alpha \in \mathcal{F}_\phi$, of $hc(\alpha)$ where hc gives the number of assertion needed to reach the minimal causal chain, as defined in Section 3.3.3.3.

This definition of the priority function f can be seen as composed of the usual $g + h$ parts where g is given by the first item while h is given by the four others. More specifically for h, the items 2 to 4 give the search effort directly visible as flaws in the partial plan while the fifth item give a heuristic estimation of the search effort to be made.

Flaw Selection. Our flaw selection strategy aims at sorting flaws in order to select the one whose resolution will be most beneficial (or least detrimental) to search. Choosing which flaw to select next is a tricky question as it mainly permits an organization of the

search space. A similar use case occurs in constraint satisfaction problems as the choice of the next variable to be given a value. A very efficient heuristic for this choice is to choose the one with the smallest domain because it results in the smallest branching factor for the early stages of search. As a result, the search space of constraint solver can be kept small. Things are more complex in planning as a plan has more types of components whose interactions are hard to take into account. Our strategy is given by the ordered list below. If the first item gives the priority to one flaw over another, then only this one is kept. Otherwise the next rule is used to break the tie.

1. Prefer a flaw that has a single resolver. A single resolver means that there is a single option to choose from. Thus, no mistake can be made in applying it.

2. Prefer unrefined tasks over other types of flaws. This priority has the advantage of giving the priority to decomposition, thus quickly reaching the point where the plan contains most of its actions with no unrefined tasks. This is useful as unrefined tasks are weakly accounted for in our plan selection strategy. Getting rid of those early means we will quickly get to a point where our f function is most informative.

3. Prefer unsupported assertions over other types of flaw. Given two unsupported assertions, choose the one on the maximally abstract state variables as defined by Knoblock [Kno94].

4. Finally, prefer the flaw with the least number of resolvers.

Note that if there is a flaw with no resolver, then this partial plan is necessarily a dead-end and the planner can select a new partial plan directly.

3.3.4.2 Forward Hierarchical Search Strategy

Our second search strategy is dedicated to HTN problems and is conceptually similar to the forward search techniques of HTN planners like SHOP2 or SIADEX. The key idea is to hand back some control to the domain designer about which plans will be explored first. For this reason, the planner will try the different decompositions in the order defined in the domain and commit to them until they are proved unsound. The planner also follows an early commitment strategy through its flaw selection strategy, which is meant to detect inconsistencies in the current plan early on.

Plan selection. Plan selection is made in a depth-first manner with chronological backtracking. When a node is expanded, the choice of the next partial plan to expand among its children is made as follows:

- if the last resolved flaw was an unrefined task, meaning that each possible partial plan matches a possible action for refining it, then give priority to the action defined first in the domain.

- otherwise sort the candidate children by the priority function of the general search strategy.

This prioritization allows the domain designer to force the planner to explore plans in a predefined order. For instance a typical prioritization for different actions to go somewhere would be to *(i)* first try to do nothing (i.e. check whether we are already at

the destination), *(ii)* otherwise try to go by foot, *(iii)* if the destination is to far, then take the car. The rest of the search decisions, such as the choice of how to resolve a threat, is let to our general search strategy.

Flaw selection. Similarly to the general search strategy, we define the flaw selection strategy as a sequence of rules whose application is meant to give the next flaw to resolve:

1. Prefer flaws with a single resolver.

2. Prefer the flaw that has the earliest interaction time. The interaction time of an unsupported assertion (resp. an unrefined task) is defined as the earliest time of its start time point (e.g. $dist_{STN}(\mathcal{O}, t_s)$ for an unsupported persistence $\langle [t_s, t_e]\, sv = v \rangle$). The interaction time of a threat is taken as the maximum of the interaction time of both assertions it involves.

3. Prefer threats over other types of flaws.

4. Prefer unrefined tasks over other types of flaws. When comparing two unrefined tasks, prioritize the one that was introduced first.

5. Like for the general strategy, prefer unsupported assertions and prioritize among them the maximally abstract ones.

6. Prefer flaws with the least number of resolvers.

The general idea behind those rules is to bias the planner into a forward search by dealing with unsupported assertions and unrefined tasks that appear early in the partial plan. Indeed, solving them will typically result in the introduction of causal chains involving assertions from the initial state. The construction of those causal chains forces the instantiation of object variables involved in them and permits an easier verification of the conditions on actions.

While this is the general idea, it does not prevent the planner from inserting some of the last actions of the plan early during search by solving a flaw with a single resolver.

3.4 Related Work

3.4.1 PDDL Temporal Planners

Ever since the introduction of the durative actions of PDDL2.1 for the third IPC, the PDDL ecosystem has seen the development of many temporal planners, a good part of which have been participating in the competition. Mainly because of the language barrier, the IPC community has been a relatively closed environment with no timeline based planners participating in the competition (with the exception of IxTeT in the IPC-2002). We here give a brief overview of the temporal planning systems that have been involved in the IPC, with some exceptions to include closely related planners. An empirical comparison with such planners will be given in Section 3.5.

Forward-chaining planners. Not surprisingly, most temporal planners participating in the IPC more or less directly descend from classical planners. And, as for the classical planning tracks of the competition, the temporal satisficing track has been dominated by forward-chaining temporal planners. We partition forward-search planners into three categories depending on their search space:

- *First-fit temporal planners* are disguised classical planners that essentially reschedule a sequential solution. The most surprising example is the baseline planner that (unofficially) won the temporal track of IPC-2008 [HDR08]. Implemented by the organizers to provide basic results to compare against, it greedily rescheduled the sequential solution provided by MetricFF [Hof03]. A more advanced implementation of this approach is YAHSP [Vid04; Vid11b; Vid14] whose second and third versions respectively won the temporal tracks of IPC-2011 and IPC-2014. This rescheduling approach is also taken by other kinds of classical planners to support temporally simple problems (e.g. MIPS [Ede03] and an early version of LPG [GSS06]).

  While such planners have the advantage of being simple, they are incomplete as they can only solve temporally simple problems that do not require concurrency between actions [Cus+07a].

- Another possibility that was first proposed in TLPlan [BK00], is the one of *decision epoch* planning. In this setting the planners have a timestamp (called the decision epoch) at which they can schedule the actions. Successors of search nodes are generated by either starting a new action at the timestamp or advancing the timestamp (typically to be just after the next effect). This technique has been the base of many influential temporal planners such as SAPA [DK03], Temporal Fast-Downward (TFD) [EMR12], TP4 [HG01], among others [Has06; BK00].

  Such planners support some cases of required concurrency but are still not complete for temporally expressive problems [Cus+07a]. Much like first-fit planners, they suffer from a lack of systematicity because they choose when actions can be executed before what to execute. To see why this can by problematic, consider an action that is entirely independent of the rest of the plan. The planner will consider its execution at each decision epoch, leading to as many plans that should have been seen as equivalent.

- A last approach that avoids those limitations is the one of *temporally lifted planners* whose line started with CRIKEY [HLF04]. Such planners essentially separate the problem of what actions to add to the plan and of when to execute them by having an STN that keeps track of temporal constraints. Those ideas have been perpetuated in the many successors of CRIKEY: CRIKEY3 [Col+08], POPF [Col+10], COLIN [Col+12] and OPTIC [BCC12].

  Unlike First-Fit and Decision Epoch planners, those temporally lifted planners are complete for the semantics of PDDL2.1 and can solve problems with required concurrency or interdependent actions [Cus+07a].

Regardless of their search space, all forward-chaining planners rely on heuristics to guide their exploration. Not surprisingly, those heuristic are mainly adapted versions of one that have been successful in classical planners. Much of those heuristic are based on a (temporal) relaxed planning graph (RPG) built by ignoring the delete effects of

actions. YAHSP uses the h^{add} heuristic by Bonet and Geffner [BG01] that essentially adds the depth of all goals fluents in the RPG built from the current state. TFD uses an adaptation of the *context-enhanced additive heuristic* h^{cea} [HG08] of the classical Fast-Downward planner [Hel06]. h^{cea} is an generalization of h^{add} to take into the context in which the transitions taken occurred. SAPA, CRIKEY, CRIKEY3, POPF, COLIN and OPTIC all use as a heuristic the length of a relaxed plan extracted from a TRPG. This heuristic was first used as h^{FF} by the classical planner FF [HN01]. TP4 [HG01] and HSP^*_a [Has06] both use a variant of the h^m heuristic that was first defined for classical planning by Haslum and Geffner [HG00] as a generalization of the h^{max} heuristic defined for HSP [BG99].

While all those heuristics have proved extremely useful for those planners, their effective adaptation to our setting is far from trivial. Indeed, their efficiency relies on a fully defined state for which we want to find the distance to fully defined set of goals. Our least-commitment approach with partial orders and lifted actions makes them hardly applicable to our case.

Other notable approaches. A pioneer work was done by Smith and Weld [SW99] in their temporal extension of GraphPlan [BF97]. Their planner, TGP, considers durative-STRIPS actions and finds a solution by extracting it from a planning graph. In the same line, LPGP by Long and Fox [LF03] is a first complete approach to PDDL2.1. It decouples the causal parts of the problem, dealt with in a GraphPlan framework, and the temporal parts that are handed to a linear programming solver. TLP-GP [MR08] does a similar work but uses a disjunctive temporal network instead of the linear program.

LPG [GSS03; GSS06] is a planner that builds an action graph (with similarities to planning graphs and to the partial plans of plan-space planning) through stochastic local search. Its latest version is able to handle problems with required concurrency by splitting durative actions into instantaneous ones and simultaneously considering temporal constraints in an STN [GSS10].

Plan-space planning has been represented by VHPOP [YS03], a ground plan-space planner that uses an adapted h^{add} heuristic to guide itself in the set of ground partial plans. CPT by Vidal and Geffner [VG06] is a more involved ground plan-space planner that aims at finding plans with an optimal makespan. This is done by placing an upper bound on the makespan of the plan and trying to prove, through inference and search whether such a plan exists. To do so, CPT performs a lot of temporal reasoning that is relevant to our setting. It starts by propagating the earliest appearances of fluents and actions from the initial state by using the admissible h^m heuristic (with $m = 2$). This is similar to what we do with our reachability analysis except that we use the weaker h^{max} (or synonymously h^m with $m = 1$). The key to the efficiency in optimal planning of CPT is a set of pruning rules that detect when a resolver would result in an overlength plan. Those rules are encoded as a CSP and used to infer temporal constraints. Many of those rules would be applicable in our planner with some adaptation to our lifted representation. However, their efficiency in CPT is strongly tied to the presence of the upper bound on the plan length which is not transferable to a non-optimal planner such as FAPE. Furthermore, to facilitate reasoning, CPT makes the restrictive assumption that interfering actions do not overlap and is thus not complete.

Planning as satisfiability have been historically strong in optimal classical and parallel planning (e.g. with planners such as SASE [HCZ10] or SATPlan [KSH06]) but few work as considered the extension to temporal planning. The recent proposal of ITSAT [RG15]

closes this gap with a temporal planner supporting PDDL2.1 problems with required concurrency. This is achieved by first solving, in a SAT solver, an non-temporal problem where all durative actions have been split into instantaneous ones. The planner then tries to find a schedule for this plan by considering the temporal constraints in an STN. If no such schedule exists, the problem is extended with an additional clauses forbiding the cause of the temporal inconsistency and the procedure is restarted.

Discussion. We first note that all reviewed planning systems strongly rely on a ground representation both for their representation of a plan during search and for computing their heuristics. We remark that this makes harder the adaptation of the wealth of heuristics that have been developed for those planners to the lifted representation of our planner. On a more general note, this raises the question of whether those planning system can be adapted to a more general setting where problems might contain thousands of objects, leading to millions of ground actions instances. There has recently been some tentative to lift this limitation for classical planners by, e.g., Ridder and Fox [RF14] construct a lifted planning graph for use in heuristic evaluation and [WSH16] tries to lift the search space of a classical planner by compiling symmetries in the domain definition. However their methods are still incremental developments that do not depart from the other limitations of classical planning and have not been adapted to temporal planning yet. The approach we take in FAPE is more viable as it can reason on entirely lifted problems and only grounds the actions to better guide the search.

We also remark that, in the planners we have reviewed, the only one supporting PDDL2.2 is OPTIC (together with a subset of PDDL3) all others planners being limited to PDDL2.1. This greatly limits the expressivity of those planners as PDDL2.1 does not allow expressing problems with timed initial literals or deadlines. While such features can be compiled into PDDL2.1 (see [FLH04]), doing so makes a tight integration of a planner into a more general acting system harder as it requires to translate the problem back and force between its natural representation and the one of the planner. Other than OPTIC, a notable exception is TLP-GP that uses a more expressive language with timed initial literals, deadlines and even conditions and effects at arbitrary fixed times within an action.

Regarding planners complete for the semantics of PDDL2.1 (i.e. CRIKEY(3), POPF, COLIN, OPTIC, VHPOP, LPG, LPGP, TLP-GP, ITSAT), we remark that they all use some kind of temporal lifting and use temporal networks or linear programming to represent their temporal constraints. This allows them to obtain flexible partial plans which is desirable for execution. We further remark that, of those complete planners, all but VHPOP and TLP-GP require splitting the action into as many instantaneous action as there are timepoints in the original action. While this is doable (with added complexity) for PDDL where all actions are limited to timepoints, adapting it to a language with an arbitrary number of timepoints is at best non-trivial and might lead to important performance loss.

3.4.2 Hierarchical Planners

HTN planning has historically been developed around a plan-space approach with planners such as NOAH [Sac75], Nonlin [MR91], SIPE [Wil90; WM95], O-Plan [TDK94] and UMCP [EHN94]. At a time where heuristics were in their infancy in automated planning, hierarchies allowed increasing the planners' performances by guiding their search. Much

like plan-space planners, those planners maintained a partial plan representation with causal links and further allowed the transformation of an abstract action into subactions through decomposition. While there is a lack of general evaluation of the performance of those planners, the general survey by Georgievski and Aiello [GA15] tends to indicate that they have difficulties in scaling up to problems handled by today's state of the art domain independent planners.

A major shift in hierarchical planning occurred with the introduction of state-based HTN planners by SHOP [Nau+00] and SHOP2 [Nau+03]. SHOP2 follows a forward chaining mechanism by sequentially building a plan. For this purpose, SHOP2 *nondeterministically* chooses the next task to consider among the ones that have no predecessors. If the task is abstract, it then branches on the choice of a method to decompose it. Otherwise, the corresponding primitive action is appended to the current sequential plan. Much like state space planners, this scheme allows the maintenance of an explicit current state against which all planning decisions can be evaluated. This is primarily used to check which methods are applicable: a method is only applicable if all its preconditions are contained in the current state. This allows early pruning without impacting the completeness of the approach: indeed a missing precondition cannot be achieved since the addition of any action in the plan would occur after the current state and thus cannot serve as an achiever. In comparison, our plan-space approach cannot make similar pruning since additional actions might appear between the task currently being refined and the temporal origin. However, this scheme has an important drawback: when a set of actions can be maintained partially ordered, SHOP2 will consider all their possible sequences. SHOP2 also uses the state as a base to define some domain dependent search control, e.g., one can define state-dependent priority functions for the order in which values of a parameter will be tried. In general the full state can be used as a base to control all aspects of the search, starting from which task to first decompose and which method to try first down to the instantiation of all parameters of the actions. This provides tremendous power to the domain designer to provide guidance to the planner as well as domain specific procedures for computing the current state. The downside side is that more stress is placed on the domain designer as SHOP exclusively relies on those search control mechanisms.

While SHOP and SHOP2 have seen some simple extension for handling time [YN00; Nau+03; Gol06] the most elaborated temporal state-based HTN planner is SIADEX [Cas+06; Fde+06]. SIADEX allows the placement of effects at arbitrary timepoints within durative actions. Conditions are restricted to be placed at the action's start. Like SHOP2, SIADEX builds an inherently sequential solution through action chaining. The main change is that SIADEX performs an online scheduling of the plan by essentially constraining an action to start after all its preconditions are true. This last step is done by keeping track of temporal constrains in an STN with propagation techniques tailored for the specificities of HTN planning and allows for the representation of partially ordered plans during search [Cas+06]. However, if multiple sequential solutions would lead to the same rescheduled plan, the partially ordered plan would appear as many times in the search space. With this regard, SIADEX has the same limitations as SHOP2. The maintenance of temporal information at planning time is nevertheless needed to reason on deadline and temporal synchronization between tasks (as supported by their problem definition).

An alternate approach to hierarchical planning has been proposed in the form of Hier-

archical Goal Networks (HGN) [Shi+12]. Shivashankar et al. replace the task networks of methods with totally ordered goal networks. Search proceeds by either *(i)* appending an action to the plan if it is applicable in the current state and achieves the first goal of the goal network; or *(ii)* extending the goal network if there is a method applicable in the current state that is *relevant* for the first goal of the goal network (i.e. the last goal appearing in the method's goal network is the next one to be achieved). The main limitation of the formalism is the restriction to totally ordered goal networks. It was partially lifted by Alford et al. [Alf+16] who allow partially ordered goal networks while still looking for a totally ordered plan (however no implementation nor empirical evaluation was provided for this extension). The HGN algorithm was recently extended to support cost-optimal planning by adapting classical planning heuristics to evaluate the minimal number of actions needed to achieve all current goals [Shi+16]. Like HTN, HGN typically requires complete hierarchical knowledge: the method should allow decomposing the initial goal network in a totally ordered sequence $\langle g_1, \ldots, g_n \rangle$ such that at most one action is needed to achieve the goal g_i from the state s_{i-1} where g_{i-1} was achieved (where s_0 is the initial state). GoDeL [Shi+13] is an interesting extension that allows a partial hierarchical knowledge that would result in "gaps" in the goal networks. Those gaps are filled either by inferring new goals through landmark regression (e.g. the technique used by LAMA [RW10]) or by using a classical planner to find a sequence of actions reaching the next goal. Their experiments show that even partial hierarchical knowledge is extremely beneficial for search while reducing the burden of the domain designer.

The use of HTN planning with task insertion was first explored with PANDA [Sch09] a lifted hierarchical planner reasoning in plan space. PANDA allows the use of high-level actions that can be decomposed into lower-level ones. Both high-level and primitive actions are always considered *free* and can thus be freely inserted by the planner to resolve open goals, outside of any decomposition tree. Parameters of the partial plan are kept lifted and handled in a binding constraint network. PANDA only supports limited qualitative time with its plan-space representation allowing it to represent partially ordered plans. Many heuristics have been studied to be used with PANDA both for flaw ordering and plan selection in a best first search setting [Sch09; Elk+12; BKB14]. More specifically, Elkawkagy et al. [Elk+12] extract necessary conditions that must be achieved in all possible decompositions of a task network and counts them in their heuristic based on the number of flaw. Similarly, Bercher, Keen, and Biundo [BKB14] estimate the minimal number of flaws that would result from decomposing the current task network. Both heuristics would be relevant for usage in FAPE as a means to account for the difficulty of decomposing the current tasks. Their effect on the scalability of PANDA is however disappointing and does not allow the planner to handle complex plans.[1]

HiPOP is a recent planner that mixes HTN and generative planning in plan-space [Bec+14; Bec16]. HiPOP considers primitive actions in the form of PDDL2.1 operators and *abstract actions* that can be transformed into a partially ordered set of primitive or abstract actions through the application of methods. An abstract action is manually annotated with effects that reflect either the ones that will be inserted by all its decompositions or some high level effects it achieves. For instance, an high level effect of an abstract action that is used to explore all locations of a given area would be to mark

[1] On the provided test data, Breadth-First Search is slightly slower but overall competitive with a Best-First search guided by the proposed heuristics.

the area explored (while its subactions would mark all locations in the area as explored). The planner considers three types of flaws: threats, unrefined abstract actions and open goals. Unlike FAPE, the introduction of a new abstract action to support an open goal is limited to actions that can support it through an effect explicitly stated in the abstract action. Because those effects are a subset of the possible effects of all decompositions, the completeness of the planner is not guaranteed. Another possibility provided by HiPOP is to manually distinguish the primary effects of an action (abstract or primitive) from its side effects. An action can only be inserted to support an open goal through its primary effects, again impacting the completeness of the approach. For both extensions, the presence of a solution plan in the search space of the planner depends on the flaw ordering heuristic and the one proposed does not guarantee completeness in general. In practice its completeness is problem dependent: for a given domain, a different formulation of the goals could make it miss otherwise acceptable solutions. HiPOP allows to give further indications to the planner: one can specify a list of predicates that an abstract action would be conflicting with and also specify which causal links should be inserted between the subactions of a method. On a general note, HiPOP provides many tools that empower the domain designer to restrict the search space. This allows HiPOP to obtain good performance on realistic robotics exploration scenarios [Bec16]. The down side is that achieving this performance requires an important amount of problem-specific knowledge rather than domain-dependent knowledge (e.g. in the encoding of an exploration domain each area to explore has a dedicated abstract action [Bec16, Annex D]).

3.4.3 Timeline-based Planners

The work on temporal planning largely predates the introduction of durative actions in PDDL2.1. Indeed, the observation by Vere, that the Partial-Order Causal Link technique can be generalized to rich temporal models, lead to numerous planners with advanced temporal representation capabilities [Ver83; GL94; Mus94; PW94; FJ03]. Vere's Deviser planner applied a plan-space approach, based on flaw detection and resolution, to a model where actions were associated to temporal intervals and problems could feature temporally extended goals and execution windows [Ver83]. The key idea in Deviser was the maintenance of the metric relationships between temporal intervals in a constraint satisfaction problem. ZENO [PW94; Pen93] extends this approach to allow reasoning on resources with continuous change. This notably includes specific data structures to keep track of assertions on the evolution and the requirements of resources and allow detection of inconsistencies in their usage as additional flaws and resolvers.

IxTeT [GL94] is a related least-commitment planner supporting explicit time and resources in a plan-space approach. Unlike the previous Zeno and Deviser, IxTeT uses timepoints as the building block for representing temporal relations. Once again a large part of the internal representation and reasoning is handled by specific constraint satisfaction problems representing constraints on timepoints and parameters of actions. IxTeT has a domain-independent search strategy based on an extended notion of least-commitment. Specifically, given a flaw to resolve, it will try first the resolver that least restricts the set of possible solutions. The least restricting resolver is heuristically chosen by measuring its impact on the underlying constraint networks (e.g. the reduction of the domain of a variable when placing an equality constraint).

Another line of work emerged with HSTS [Mus94] from the objective of tightly integrating planning and scheduling. Instead of actions, HSTS relied on the notion of compatibilities to describe the possible interactions between various timelines. For instance, on a timeline representing the position of a rover, the temporal interval (or *token* in their formalism) associated with the value `going(a,b)` must be met by a token `at(a)` and must meet a token `at(b)`. The compatibility could be used to express requirements on other timelines, e.g., that the battery of the rover be sufficiently charged for the duration of the action. The planner's objective is to find fully defined timelines that respect all compatibilities. Despite the difference in the representation, this was again achieved by a flaw/resolvers mechanism that borrowed most of its search technique from plan space planning, e.g., instead of choosing which action to use as a supporter, HSTS would choose which compatibility to enforce and introduce the necessary tokens and constraints for its implementation. HSTS was notably used as a base for the Remote Agent Planner (RAP or RAX-PS) that was demonstrated by running on board the Deep Space One spacecraft and controlling its operations during two days in 1999 [Nay+99]. The ideas of HSTS later matured into EUROPA [FJ03; Bar+12] whose central paradigm is to see planning as a dynamic constraint satisfaction problem where choices of the planner simply results in the addition of constraints to underlying constraints networks. As a result EUROPA was not only used for planning but also to handle problems that are usually reserved to CSP or SAT solvers such as scheduling and optimization problems.

Let us point out some of the keys differences between the search space of FAPE and the one of EUROPA. The core component in EUROPA is the timeline that generalizes state variables to also represent activities (or actions in our formalism). A timeline must eventually be filled with a sequence of tokens, each of which giving the value taken by the timeline over a given temporal interval. To make a direct comparison with our model, a timeline typically represents the evolution of a state variable, e.g., in our example problem we would have a timeline for the state variable $\texttt{loc}(\texttt{r1})$ and one for the state variable $\texttt{loc}(\texttt{r2})$. Tokens are best thought of as persistence assertions with lifted parameters and associated to a temporal interval. A key difference between FAPE and EUROPA appears in the way *unsupported assertions* (or free tokens in EUROPA terminology) are handled. Indeed, given a free token, EUROPA will first assign it to a given timeline. Put into FAPE formalism, this is roughly equivalent to binding all the parameters of the state variable of the unsupported assertion.

Example 3.5. Consider the simple case where we have two rovers `r1` and `r2`, initially in location `d0`, and that our objective is to have a rover in `d1`. This goal is represented by the persistence condition $\langle [t_s, t_e]\, loc(r) = \texttt{d1} \rangle$.

In EUROPA handling this unsupported assertion (or free token) would first require assigning it to a timeline, i.e., binding the r variable. The planner would then decide how to support it. In this case it would mean triggering the compatibility corresponding to $\texttt{move}(\texttt{r1}, x, \texttt{d1})$ or the one corresponding to $\texttt{move}(\texttt{r2}, x, \texttt{d1})$ depending on the choice previously made for r.

In contrast, FAPE would directly consider the insertion of a $\texttt{move}(r, x, \texttt{d1})$ action, leaving the r variable unbound. As a consequence, the search space of FAPE would be more compact, representing both possibilities in a single partial plan. On the down side, it makes automated reasoning on a partial plan more complex since it has more possible instantiations.

EUROPA was not developed as a domain-independent planner but instead provides facilities for the definition of domain-dependent search control. This typically takes the form of ordering functions to order the different flaws and resolvers. Combined with a depth-first strategy, it allows a good control on the search of the planner. The natural support for temporally rich action models further allows for a hierarchization of the search space by the introduction of abstract state variable. Indeed, mimicking our m2-transport action of Figure 2.2, one could have high-level transport activity affecting an abstract *transported* timeline. The compatibility associated with transport would in turn require the presence of load, move and unload activities by placing tokens on lower level timelines. This is however a limited form of HTN planning, as the notion of task dependency is missing and relying on state variable instead of tasks means that one is restricted to the use of task sharing (indeed a single "effect" can support multiple "conditions").

Some work has been conducted to transpose ideas from domain-independent planners into EUROPA [BS07; BS08; BS09]. The key to this work is the adaptation of Domain Transition Graphs to the internal model of EUROPA. Those are used in fashion similar to ours (Section 3.3.3.3): to estimate the difficulty of building a causal linking assertions. The main difference in the usage of this information is that Bernardini and Smith use it to estimate the "cost" of placing a given token at a given position in a timeline, giving a local view of the cost of using a specific resolver. When used to compare it with the cost of placing this token elsewhere, it gives an ordering function for the choice of a resolver. Combined with the Depth-First search of EUROPA, this heuristic proves useful in a few domains that EUROPA was able to handle without any domain dependent knowledge [BS08]. However the Depth-First strategy of EUROPA proved limiting in this setting as it places additional pressure on the heuristic: the planner could only recover from a mistake if it proved all resulting plans to be invalid. As a result, when such a mistake is made early during search, the planner would fail to find a solution as it would stay stuck in an unfeasible branch.

Much work involving planning and scheduling for space relies on timelines as the central representation (as surveyed by Chien et al. [Chi+12]). As a result, the European Space Agency (ESA) launched the Advanced Planning and Scheduling Initiative (APSI) that resulted in the definition of the Timeline-based Representation Framework (APSI-TRF) [Ces+09]. While not a planner per se, APSI-TRF aims at being a timeline-based deliberation layer to provide facilities for the implementation of timeline based planners. Those notably include a module for the representation of timelines together with specialized constraint solver to reason on temporal and binding constraints.

It has notably been used as a building block for the GOAC-APSI planner [Fra+11]. GOAC-APSI (and its predecessor OMPS [CFP08]) share most of its search mechanism with EUROPA and tries to iteratively fix flaws in a depth first search.

It has also been used as the base layer in MrSPOCK [Ces+09], a long term planner for the Mars Express mission. MrSPOCK works by first greedily constructing a long term plan that is then optimized through a genetic algorithm. One of the key insights in this work is the use of a timeline representation not only as a book-keeping structure for the genetic optimization phase but also for propagation of temporal and binding constraints in this phase.

A similar approach is taken in the meta-CSP framework that sees a planning and schedul-

ing problem as a higher level constraint satisfaction problem that requires cross reasoning on several lower level CSPs (referred to as ground CSPs). More specifically meta-constraints enforce high-level requirements on the solution plan, playing the role of the flaw detection function of our approach. A conflict associated to a meta-constraint is denoted by a *meta-variable* whose domain is a set of *meta-values*. For instance, given a *meta-constraint* enforcing the overall consistency of the plan, a *meta-variable* would be associated with each pair of conflicting assertions. The *meta-values* in its domain would be a set of separation constraints spanning both on a temporal ground CSP and on a bindings ground CSP. The key idea is to leverage this generic approach to allow a natural integration with more application specific components by the addition of supplementary ground CSPs and meta-constraints.

This general approach has been instantiated in several planners targeting robotic applications while relying on a common core for reasoning on time, causality and resource consumption. These core capabilities are extended by Di Rocco, Pecora, and Saffiotti [DPS13] to allow reasoning on information dependencies in robotic tasks. For this purpose, the activities of the robots are annotated with the information they need and the one they produce while the relationship between different informations is encoded as a set of Horn clauses. Meta-constraints enforce the production of all (directly or indirectly) needed information by the scheduling of additional activities. On another line, Mansouri and Pecora [MP14] provide an additional ground CSP for spatial reasoning, allowing them to reason on the relative placement of objects for service robot tasks.

The meta-CSP framework has recently been used as a base to create CHIMP: a (pure) HTN planner with a timeline-based representation [Sto+15]. Like traditional HTN planners, CHIMP uses a notion of task that can be decomposed into partially ordered task through the application of methods. Conditions and effects are represented as tokens that will be placed on a timeline. The former must be unified with an existing token while the latter are added to the timeline. CHIMP follows a pure HTN approach: methods and actions can only be inserted if they refine an existing task, i.e., all actions in the plan derive from the initial task network. While their input language differs, FAPE (when restricted to pure HTN planning) and CHIMP have essentially the same model with multi-valued state variables and expressive time representation that are mapped into a lifted partial plan representation whose most aspects are handled through constraint networks. The key difference between them lies in the way they explore their search space. While it is generally more flexible, CHIMP can be seen as a forward search planner: given a method or an action to insert in the plan, CHIMP immediately unifies all its conditions with existing tokens on timelines. In practice this means that all the effects supporting an action must already be in the plan when it is introduced. Because the supporting effects could be introduced by decomposing a yet unrefined task, maintaining completeness requires making the choice of which task to refine nondeterministic (i.e. a backtrack point). On the other hand, FAPE treats conditions of actions (high-level or primitive) as open goals and thus does not require branching on the choice of which task to decompose. The approach taken by CHIMP has some clear advantages. First, the fact that the planner never maintains open conditions greatly simplifies the underlying implementation. Second, the conditions on methods can be checked right away and be used to prune the search space early on by discarding non-applicable ones. The down sides are also important: *(i)* it can drastically impact the systematicity of the planner as equivalent plans can be explored multiple times, *(ii)* it forces an early commitment as unifying the con-

ditions would typically result in instantiating their parameters. Another limitation, less relevant here, is that CHIMP cannot solve problems with causally interdependent actions (see Example 2.4). To mitigate the impact on systematicity, we can make the following observation: if the actions derivable from a task t_1 have no effect that can support an action derivable from a task t_2, then t_2 can always be decomposed first without impacting completeness. Such refinement order can be explicitly enforced in CHIMP: the planner would thus try to decompose t_1 before t_2, if this order fails, then the planner would not try the other way around. The application of such enforcement is however limited: any two tasks that have no predefined order and might use a common resource cannot benefit from it. Since FAPE and CHIMP take very different approaches to reason on essentially equivalent models, we will further empirically evaluate the efficiency of their techniques in Section 3.5.3.

ASPEN is another timeline based planner proposed by the NASA Jet Propulsion Laboratory (JPL) [Fuk+97; Smi+98; Chi+00a; Chi+00b]. It shares with the others the same underlying structure of timelines whose temporal parameters are handled in a dedicated constraint network. The main difference lies in the use of iterative repair [Zwe+93] to perform local search in place of the depth-first search adopted by other timeline-based planners. This local search is driven by the resolution of conflicts in the partial plan, e.g. temporal inconsistency or lack of causal support. Due to the nature of its search, conflict resolution allows both constructive resolvers (e.g. scheduling a new activity) and destructive ones (e.g. removing an activity). This local search has practical benefits as it allows to take as input any plan regardless of its flaws and inconsistencies. Efficiency is achieved by ASPEN by the definition of a hierarchical structure where activities can be refined into subactivities, allowing the planner to quickly bootstrap its search with a minimal (possibly flawed) plan. However, this hierarchical structure strongly differs from the one of hierarchical planners as it is not constraining: some constraints due to a decomposition could be relaxed through local search. ASPEN uses hand written rules to weight the different flaws and resolvers. The search algorithm then randomly selects the next flaw-resolver pair to apply with the probability distribution favoring higher weighted elements. Unlike other timeline based planners, ASPEN tends to favor early commitment and grounds all non-temporal parameters. While increasing the search space, it makes reasoning on partial plan easier and allows for better integration with external components that do not naturally handle a lifted representation [Chi+98].

Several of these timeline based planning system have been successfully deployed in the many space applications for which they were developed (as surveyed by Chien et al. [Chi+12]). More recently, they have been adapted to other application domains including marine exploration [McG+08] and robotics [DPS13; Pec+12; Cir+14; MP14; Sto+15]. The key to this success has been the flexible timeline-based representation that allows for many extensions to the basic planning paradigm. The most prevalent such extension is the handling of resources, supported by all the above planners, that is naturally handled by dedicated flaws and resolvers. The down side of these timeline based planners is the harder integration of domain-independent search control and heuristics in particular. The need for heuristic should however be mitigated as the conflict directed search takes advantage of the structure of the problem by only extending the plan to fix a visible conflict. Furthermore the least-commitment approach allows more compact search spaces where many possible plans are represented in a single search node. On the other hand, domain-

independent planners based on state space search are extremely dependent on the quality of their heuristic because it is used to justify the choice of actions whose contribution to the overall plan is yet unknown. This strong dependency on heuristics makes them harder to extend with additional constraints since even minor changes to the allowed set of solution must be reflected on the heuristic value. Least commitment timeline-based planners have been easier to adapt to various settings due to their more general constraint based representation and their lower dependence on domain-independent heuristics.

FAPE seeks a middle ground between those two approaches by aiming for a good domain-independent performance on a restricted set of problems while maintaining the overall benefits of a timeline representation. Indeed, FAPE focuses on efficiently handling causal relationships that are the core of many planning applications. On the other hand, we acknowledge the need for more targeted extensions to solve real world problem. Two such extensions, for handling temporal uncertainty and for the maintenance of abstract long term plans, are developed in the subsequent chapters. The combination of a flexible timeline representation and of a least commitment search is key in integrating those extensions through an alteration of flaw detection functions. Furthermore, the low dependency of FAPE on heuristic values remains important to handle such cases with domain-independent search. However, when this is not sufficient, the planner can rely on domain dependent knowledge, typically provided by adding hierarchical features to the domain.

3.5 Empirical Evaluation

We have presented an algorithm able to plan both in a generative and a hierarchical fashion with rich temporal models. Our proposal is complemented with a number of techniques intended to improve the efficiency of the planner by *(i)* inferring constraints, *(ii)* guiding the planner to efficiently explore its search space.

In this section, we first compare FAPE with state-of-the-art temporal planners from the IPC. We show FAPE to be competitive in a fully generative setting and the addition of hierarchical knowledge to further improve its performance. We then study how each of the techniques we have described contribute to the overall efficiency of the system. Last, we study the impact of the respective techniques of FAPE and CHIMP to reason on their very similar models.

3.5.1 Empirical Comparison with IPC Planners

Experimental setup. For comparison with state of the art PDDL temporal planners, we consider 11 temporal domains from the International Planning Competition all of which with PDDL2.1 versions. We have manually written ANML versions of the domains that closely mirror the original PDDL model: domains have the same actions and a direct mapping from predicates to corresponding state variables. For each domain, we wrote a, domain-specific, automated translator that parsed the original PDDL problems and output ANML problem files.

We use hierarchical versions of a subset of those domains (blocks, hiking, logistics, turnandopen). Those domains were chosen either because they have a natural expression with partial hierarchies (blocks and logistics) or because FAPE had difficulties in solving the generative versions of the problem (hiking and turnandopen).

Three of the selected domains have "advanced" temporal features. Namely, *airport-tw* and *satellite-tw* have temporal windows that respectively restrict the instants at which a plane can take off and at which a satellite can transmit data. In addition, the goals of *pipesworld-dl* are associated with deadlines that must be met by the solution plan. We use compilations of those domains into PDDL2.1. All domains, in their ANML and PDDL versions, are available online in FAPE's public repository.[2]

Planners. We compare FAPE to POPF [Col+10], OPTIC [BCC12] and Temporal Fast-Downward (TFD) [EMR12]). POPF is a complete PDDL2.1 planner based on temporally-lifted progression planning [Cus+07a]. As such it can be seen as a forward-search planner taking a late-commitment approach in the ordering of actions and uses the FF heuristic adapted for temporal planning. OPTIC is a recent extension to POPF that supports more advanced PDDL features, including PDDL2.2 timed initial literals and PDDL3 preferences.

Temporal Fast-Downward (TFD) is a temporal extension of the successful Fast-Downward classical planner using a decision-epoch mechanism [Cus+07a]. It performs heuristic search in the space of time-stamped states, using an adapted version of the context-enhanced additive heuristic [HG08]. TFD supports PDDL2.1 syntax but is not complete as it only supports a limited class of problems with required concurrency. More specifically it cannot handle problems with interdependent actions.

POPF and TFD have been runners-up in the temporal satisficing track of IPC-2011 and IPC-2014 respectively. We did not consider YAHSP, the winner of those two tracks, in our comparison as it only supports temporally simple problems and as such is strictly less expressive that the other planners considered here.

We distinguish two versions of FAPE. FAPE-Gen denotes the purely generative version of FAPE that only considered flat domain encoding with no hierarchical information. It uses the general search strategy and has no domain-dependent knowledge. FAPE-Hier uses the hierarchical versions of the *blocks*, *hiking*, *logistics* and *turnandopen* domains together with the *forward hierarchical* search strategy. For flat domains, it uses the general search strategy and is equivalent to FAPE-Gen.

Results. All tests were performed on an Intel Core i7 and allowed to run for 30 minutes with 3GB of RAM. The results are given in Table 3.2 in terms of the number of problems solved by each planner within the time limit. POPF crashed while preprocessing two domains (*pipesworld-dl* and *satellite-tw*). For this reason, we provide a subtotal excluding said domains to facilitate the comparison.

The performance of the purely generative version of FAPE is comparable with that of POPF and OPTIC. TFD is ahead in terms of number of problems solved. The addition of hierarchical knowledge in 4 of the domains allows FAPE to solve 36 more instances. As a result, it outperforms POPF and OPTIC and but still falls behind TFD in terms of solved problems.

A focused subset of the results is given in Table 3.3 for problems with deadlines and timewindows. Those are the only domains of the test set where time is strictly needed, i.e., on all other domains every solution plan has a valid totally ordered counter part. While being the overall best performer, TFD exhibits poor performance on those domains, solving only 3 problems.

[2]Available at `https://github.com/laas/fape`

	FAPE-Gen	FAPE-Hier	POPF	OPTIC	TFD
(ipc4) airport	6	6	7	7	**37**
(ipc4) airport-tw	7	7	**17**	7	1
(ipc2) blocks*	25	27	32	32	**35**
(ipc8) driverlog	**4**	**4**	0	0	0
(ipc2) logistics*	22	**27**	**27**	**27**	**27**
(ipc4) pipesworld-dl	6	6	-	**13**	2
(ipc5) rovers	**34**	**34**	26	26	29
(ipc4) satellite-tw	**10**	**10**	-	4	0
(ipc8) satellite	16	16	3	4	**17**
(ipc8) turnandopen*	0	8	8	9	**18**
(ipc8) hiking*	0	**20**	10	9	19
Subtotal (POPF specific)	114	149	130	121	**183**
Total	130	165	-	138	**185**

Table 3.2: Number of problems solved in 30 minutes for various temporal IPC domains. The best performance in given in bold. FAPE-Hier uses hierarchical versions of the starred domains and generative versions of the others.

	FAPE (Gen/Hier)	OPTIC	TFD
(ipc4) airport-tw	**7**	**7**	1
(ipc4) pipesworld-dl	6	**13**	2
(ipc4) satellite-tw	**10**	4	0
Total	23	**24**	3

Table 3.3: Results limited to domains featuring deadlines or timewindows. FAPE-Hier does not appear separately as we only considered generative versions of those domains. POPF is excluded since it failed to process two of those domains.

3.5.2 Evaluation of the Different Components of the Planner

3.5.2.1 Evaluation of Reachability Analysis

We start by evaluating our proposed reachability analysis independently of other techniques. The motivation to do so comes from the fact that it generalizes the delete-free analysis done by classical planners to a temporal setting. More interestingly, the other adaptation of this technique by Coles et al. [Col+08] is easily represented in our framework by considering an additional relaxation. We can hence make a direct comparison between them.

Tested Configurations. We distinguish 5 configurations of the planner depending on how far it pushes the reachability analysis.

- R_∞ is the configuration where no limitation is put on the number of iterations for reachability analysis.

- R_5 and R_1 denote the configurations where the number of iterations is limited to 5 and 1 iterations, respectively. This makes the algorithm strongly polynomial and reduces the overhead when many iterations are needed to converge. On the other

hand, the algorithm might incorrectly label unreachable actions as reachable and fluents/actions will typically be found to have lower earliest appearance times.

- R^+ denotes the configuration where all after-conditions are ignored. In practice, it means that the propagation will stop right after the first Dijkstra propagation (in the middle of the first iteration). This configuration is equivalent, for our more expressive temporal model, to the reachability analysis performed by POPF and related planners [Col+10; Col+08; BCC12; Col+12].

- $\varnothing$ denotes the configuration where no reachability analysis is performed. In this case, the planner does not ground the problem which reduces its overhead.

Experimental setting. We evaluate our reachability analysis technique on several temporal domains with and without hierarchical features, the former involving many interdependencies between high-level actions and their subactions. The *satellite*, *rovers*, *logistics* and *hiking* domains are the eponymous domains from the International Planning Competition. The *handover* domain is a robotics problem presented by Dvorák et al. [Dvo+14b], the *docks* domain is the dock worker domain of Ghallab, Nau, and Traverso [GNT04] and *machine-shop* is the problem presented by Cushing et al. [Cus+07b]. Hierarchical versions of the domains have their names appended with '-hier'. We note that the flat versions of the domains *satellite*, *rovers* and *logistics* are temporally simple and contain no after-conditions. On such domains, our first four configurations are equivalent as discussed in Section 3.3.2.3. All experiments were conducted on an Intel Xeon E3 with 3GB of memory and a 30 minutes timeout.

	R_∞	R_5	R_1	R^+	$\varnothing$
(IPC-8) satellite (20)	14	14	14	14	**15**
(IPC-5) rovers (40)	**25**	**25**	**25**	**25**	**25**
(IPC-2) logistics (28)	**8**	**8**	**8**	**8**	**8**
(IPC-8) satellite-hier (20)	**17**	**17**	**17**	**17**	16
(IPC-5) rovers-hier (40)	**22**	**22**	**22**	**22**	**22**
(LAAS) machine-shop-hier (20)	**7**	**7**	**7**	**7**	**7**
(IPC-2) logistics-hier (28)	**28**	**28**	**28**	6	9
(LAAS) handover-hier (20)	**16**	**16**	**16**	7	7
(IPC-8) hiking-hier (20)	**20**	17	16	15	17
(LAAS) docks-hier (18)	**17**	13	12	7	7
Total (254)	**174**	167	165	128	133

Table 3.4: Number of solved tasks for various domains with a 30 minutes timeout. The best result is shown in bold. The number of problem instances is given in parenthesis.

Results and Discussion. Table 3.4 and Figure 3.10 present the number of problems solved using different reachability models. For this criteria, R_∞ outperforms the other configurations: solving the highest number of problems on all but one domain. R_5 and R_1 are respectively second and third best performers while R^+ does not provide significant pruning of the search space; the computational overhead makes it perform slightly worse than no reachability checks (denoted by $\varnothing$). As expected, on temporally simple problems (non-hierarchical domains in our test set), all configurations show similar performance.

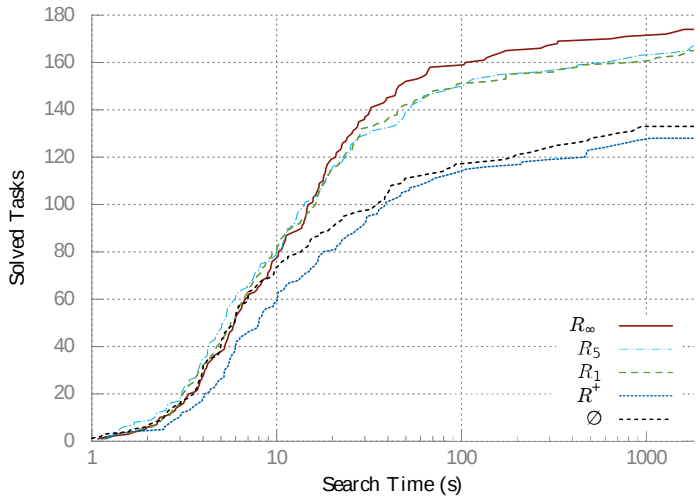


Figure 3.10: Number of solved tasks by each configuration within a given time amount.

	R_∞	R_5	R_1	R^+	∅
satellite	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
rovers	**43.5**	**43.5**	**43.5**	**43.5**	0.0
logistics	**34.5**	**34.5**	**34.5**	**34.5**	0.0
satellite-hier	**14.1**	**14.1**	**14.1**	**14.1**	0.0
rovers-hier	**72.6**	**72.6**	**72.6**	27.1	0.0
machine-shop-hier	**87.9**	**87.9**	**87.9**	0.0	0.0
logistics-hier	**94.6**	**94.6**	**94.6**	15.5	0.0
handover-hier	**99.2**	**99.2**	**99.2**	3.5	0.0
hiking-hier	**38.1**	36.5	36.5	0.0	0.0
docks-hier	**85.2**	52.6	52.6	0.0	0.0

Table 3.5: Percentage of ground actions detected as unreachable from the initial state. For each problem instance, the percentage is obtained by comparing the number of ground actions detected as unreachable from the initial state with the original number of ground actions. Those values are then averaged over all instances of a domain.

Table 3.5 presents the percentage of actions detected as unreachable by different configurations. As expected, R_∞, R_5, R_1 and R^+ perform identically on temporally simple problems. However, R^+ is largely outperformed on all but one hierarchical domain. The good performance of R_1 with respect to R^+ shows that a single iteration is often sufficient to capture most of the problematic after-conditions. However, on more complex problems such as *hiking-hier* and *docks-hier*, more iterations are beneficial both in terms of detected unreachable actions and solved problems.

	R_∞	R_5	R_1	R^+	$\varnothing$
satellite	**100** (1)	**100**	**100**	**100**	–
rovers	**100** (1)	**100**	**100**	**100**	–
logistics	**100** (1)	**100**	**100**	**100**	–
satellite-hier	**100** (2)	**100**	**100**	**100**	–
rovers-hier	**100** (4)	99.2	56.7	54.2	–
machine-shop-hier	**100** (6)	69.3	14.2	14.2	–
logistics-hier	**100** (2)	**100**	2.8	2.8	–
handover-hier	**100** (43)	98.2	5.7	5.7	–
hiking-hier	**100** (9)	**100**	71.7	71.7	–
docks-hier	**100** (37)	73.0	29.8	29.8	–

Table 3.6: Average admissible makespans for different reachability models. Those are computed by taking the earliest appearance of the latest satisfied goal from the initial state, and normalizing on the value computed for R_∞. For R_∞, we also indicate the average number of iterations needed to converge on the first propagation of each instance (in parenthesis).

Table 3.6 presents the value taken by the admissible h^{max} heuristic with different reachability models. h^{max} essentially determines a lower bound on the makespan of a solution plan by looking at the earliest appearance of the goals. On all but one hierarchical model, both R_1 and R^+ largely underestimate the makespan of a solution. Indeed, not propagating after-conditions makes them miss important causal aspects of the problems. Those can take as many as 43 iterations to be initially propagated by R_∞. The subsequent propagations are typically faster because they are made incrementally. As expected, a single iteration was needed to converge on all temporally simple problems.

3.5.2.2 Evaluation of Other Components

We continue our evaluation with a broader scope by evaluating how each of the techniques discussed contributes to the overall efficiency of the system. For this purpose we consider 6 configurations of the planner each one having a specific technique deactivated. We compare this against the full configuration of FAPE.

- $\mathcal{C}_{full}$ is the full configuration will all the techniques previously discussed enabled. It use the general search strategy by default and switches to the *forward hierarchical* strategy when facing a fully hierarchical domain.
- $\mathcal{C}_{gen}$ is the configuration where FAPE always uses the general search strategy. It differs from $\mathcal{C}_{full}$ on fully hierarchical domains where the forward hierarchical search strategy would have been preferred.

- $\mathcal{C}_{lift}$ is a fully lifted version of FAPE. More specifically it does not use reachability analysis and thus all features that require grounding are deactivated (i.e. DTGs and instantiation/refinement variables). This configuration is essential to measure the benefits and penalties for grounding the problem.
- $\mathcal{C}_{NoCN}$ does not use the causal network to infer constraints or prune impossible resolvers (Sections 3.3.3.1 and 3.3.3.2).
- $\mathcal{C}_{NoDelCheck}$ does not check that there is a sufficient delay from the start of a task to the moment its possible effect is required to support a task. Hence it can result in additional resolvers being considered for supporting an open goal.
- $\mathcal{C}_{NoDecVars}$ does not use refinement variables to disregard the resolvers of unrefined tasks that involve unreachable actions.
- $\mathcal{C}_{A^*}$ use A^* in place of our variant of A^ϵ (Section 3.3.4.1).

It should be noted that our test configurations do not consider variations for the flaw ordering strategies. We have tested other such strategies but they resulted in poor performance of the planner. While in general other strategies can be efficient (e.g. many ground state-space planners use a "threat first" strategy), their efficiency is strongly coupled with the rest of the strategies used in the planner. This factor can explain that dominance of the current strategy as it has evolved with the rest of the system.

Domains. We use a set of domains to evaluate the performance of the planner. A large part of those have been translated from the International Planning Competition: *airport*, *blocks*, *driverlog*, *logistics*, *pipesworld*, *satellite*, *turnandopen* and *hiking*. The domain definitions of those problems have been manually translated into ANML and their problems were automatically translated by domain dependent parsers.

Our other domains are the following. *docks* and *handover* are the ones already presented. *race* is a robotics domain adapted from CHIMP [Sto+15], where a waiter-robot must serve clients. It notably features navigation constraints expressed through hierarchical features and deadlines regarding the moment a client must be served. *springdoor* is another robotics problem where the robots must move objects between several places with closed doors. Opening a door results in complex interactions between several actions (turning the knob, pushing the door and releasing the knob) which can be performed by the robot that must pass through the door if it carries nothing or by another one. *machine-shop* is a domain presented by Cushing et al. [Cus+07b] of baking pieces in kilns where the action of baking must be concurrent with an action showing that the kiln is switched on.

We consider several variants of the problems depending on whether they are fully generative (denoted by a FLAT suffix), fully hierarchical (denoted by a FULLHIER suffix) or partially hierarchical (denoted by a PARTHIER suffix). In fully hierarchical domains, all actions are task-dependent while partially hierarchical domains contain some free actions.

Overview of results. We first start by giving a broad overview of the results given in Table 3.7. The table contains the number of problems solved by each configuration with a five minutes timeout and a memory limit of 3GB.

The first important catch is to see that while all features somewhat contribute to the efficiency of the planner, none is critical to its overall performance. Indeed, the absence

	$\mathcal{C}_{full}$	$\mathcal{C}_{gen}$	$\mathcal{C}_{lift}$	$\mathcal{C}_{NoCN}$	$\mathcal{C}_{NoDelCheck}$	$\mathcal{C}_{NoDecVars}$	$\mathcal{C}_{A^*}$
airport-FLAT (50)	5	5	5	5	5	5	5
airport-tw-FLAT (50)	6	6	6	6	6	6	6
blocks-FLAT (35)	23	23	26	23	23	23	23
docks-FLAT (18)	12	12	12	12	12	12	12
driverlog-FLAT (20)	3	2	1	2	2	3	0
handover-FLAT (12)	8	8	4	8	8	8	8
logistics-FLAT (27)	23	22	3	19	21	21	13
pipesworld-dl-FLAT (30)	5	5	6	5	5	5	5
rovers-FLAT (40)	32	32	35	32	32	32	31
satellite-tw-FLAT (36)	9	9	10	8	9	9	5
satellite-FLAT (20)	16	16	16	13	16	16	11
turnandopen-FLAT (20)	0	0	0	0	0	0	0
Total Generative	142	140	124	133	139	140	119
blocks-PARTHIER (35)	28	27	32	27	27	28	26
blocks-FULLHIER (35)	10	7	14	9	9	4	10
docks-FULLHIER (18)	17	17	17	17	11	15	17
handover-PARTHIER (12)	10	10	1	10	10	10	10
hiking-FULLHIER (20)	20	2	2	20	13	12	20
logistics-PARTHIER (27)	27	27	27	27	27	27	27
race-FULLHIER (13)	13	10	5	13	13	13	13
satellite-PARTHIER (20)	17	17	17	15	17	17	10
springdoor-FULLHIER (4)	2	2	2	2	2	2	2
machine-shop-PARTHIER (10)	6	6	6	5	6	6	6
turnandopen-FULLHIER (20)	8	0	0	8	5	0	8
Total Hierarchical	158	125	123	153	140	134	149
Total	300	265	247	286	279	274	268

Table 3.7: Number of problems solved by each configuration with a 5 minutes timeout.

of a feature resulted in 13 to 53 less problems being solved which is only a small subset of the overall 300 problems solved by the full configuration. However, this statement must be mitigated as the difficulty of a planning problem is typically exponential in the number of goals in the problem. Solving a few additional problems of a domain has thus more significance than it seems through a simple comparison of numbers.

As expected the $\mathcal{C}_{gen}$, $\mathcal{C}_{NoDelCheck}$ and $\mathcal{C}_{NoDecVars}$ have no impact on generative problems (the small difference is due to problems being solved closed to the timeout by one configuration). The use of our *forward hierarchical strategy* is however critical in some hierarchical domains, as it can be seen by the terrible performance of $\mathcal{C}_{gen}$ on the *hiking* and *turnandopen* domains. The use of delay checks (absent in $\mathcal{C}_{NoDelCheck}$) and of refinement variables (absent in $\mathcal{C}_{NoDecVars}$) is less critical but contributes to the scaling up of the planners on some hierarchical domains. The gain of using causal networks (absent in $\mathcal{C}_{NoCN}$) and A^ϵ (absent in $\mathcal{C}_{A^*}$) is also globally noticeable but does not benefit a domain in particular.

The most important results are the ones related to $\mathcal{C}_{lift}$ as this configuration does not ground the problem. Thus it can be expected to scale up better on problems that would feature many ground actions. The gains of non-grounding are noticeable on the *blocks*, *rovers* and *satellite* whose most difficult problems contain many objects. A limitation of our test set (and of problems from the IPC in general) is that the difficulty of the problem (e.g. length of the plan) is directly correlated to the number of objects in the problem. Thus we have no instances in our test set where $\mathcal{C}_{lift}$ would find a trivial solution plan while $\mathcal{C}_{full}$ would fail because it could not ground the problem. On the rest of the test set, we can see the gain, in terms of search control, of applying reachability analysis. This is most noticeable with the hierarchical versions of *handover*, *hiking* and *turnandopen*. Nevertheless, we believe it is important to keep the ability to perform a fully lifted search even if it means not using some of the heuristics we have developed. A view of the overhead required for grounding can be seen in Figure 3.11, where we can see that the lifted version tends to solve simple problems faster because it does not need to ground the problem.

Partial vs Full Hierarchy. A good number of domains used partial hierarchies. A good example of the benefits of using those is the *blocks*-PartHier, given in Section B.1. In this domain, the single task-dependent primitive action is stack. When compared to a flat version of the problem it features an additional high-level action DoStack(a, b) that either does nothing if b is on a or decomposes to stack(a, b) if it is not. In the resulting problem, the planner is only allowed to perform one stack action per `DoStack` task in the problem. On the other hand, it can use as many pickup, putdown and unstack as necessary to fulfill the conditions of the stack actions. This formulation thus poses a simple constraint on the plan: do not use more stack actions than necessary. When compared to the fully hierarchical version of the domain (Section B.2), the partially hierarchical one is obviously simpler requiring less domain engineering. Furthermore the partially hierarchical version is more easily solved by FAPE. This should not be too surprising: FAPE already does a decent job of solving the flat version of the problem and this simple extension simply provides some additional help. On the other hand the hierarchical version is a very different problem. While extending the fully hierarchical version with more domain knowledge could probably make it competitive with the partially hierarchical one, there is no need to do it because the simpler partially hierarchical version already fulfills our needs (and we are not eager to dedicate much time in solving blocks world problems).

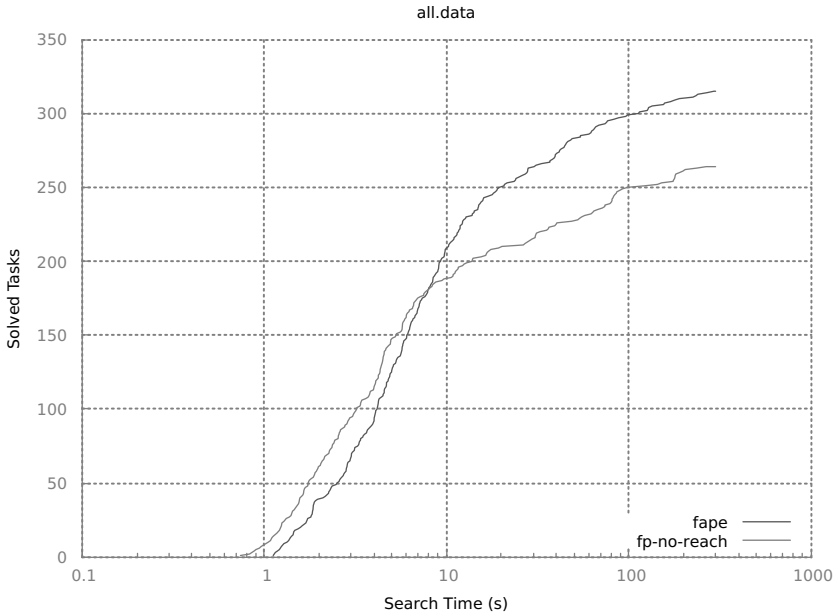


Figure 3.11: Time needed to solve problems with $\mathcal{C}_{full}$ (fape) and $\mathcal{C}_{lift}$ (fape-no-reach). The startup time of the JVM that take an important share of time to load and perform just-in-time optimization during the first seconds of the run. This overhead is however comparable for both configurations.

A similar approach is taken in the *handover*-PartHier and *logistics*-PartHier. In *handover*, the only task-dependent actions are those involving manipulation of an object (pick, place and handover) while the movements of the robots are left free. Similarly, in *logistics* the only hierarchical part are the load and unload actions. All the movements of planes and trucks are left free. If the planner was to decompose entirely the original task network, it would have a set of load and unload actions and would simply need to plan the fleet movements between those.

Specificities of fully hierarchical problems. The fully hierarchical problems (and the search strategy that comes with them) hand back some control to the domain designer and as such must be considered more carefully. The hierarchical search strategy will try to decompose a task by trying the actions in the order of their definition. To make sure this order is respected, the planner uses a depth-first search and thus commits to a refinement until it is proven to be unfeasible. This is practical for the domain designer as it ensures that its will is respected and gives him more visibility on how the planner will act.

Depth-first search is however a dangerous thing when the search space is infinite as our own. In this case, infiniteness does not come from the properties of plan space planning because there are no free actions to insert. However HTN planning does allow an infinite search space in the presence of recursive methods. In this case, it is the responsibility of the domain designer to make sure that the planner will not get stuck in an infinite recursive decomposition. There are several ways to do that: one can put a bound on the depth of the solution or a constraint on the duration of the plan. The most general (and often most efficient) solution is however to force the termination of a recursive method once it has exhausted all relevant solutions as shown in Example 3.6.

Example 3.6. Consider the two high-level actions below that recursively try to get an agent to some location *dest*. It is perfectly possible for the agent to go around in circles indefinitely. For instance, consider we have two connected places a and b and an unconnected place c where we want to get (i.e. we have a `goto`(c) task). Starting from a, the left action would never be chosen and the planner would indefinitely choose the right action that would move between a and b while always posting a new unrefined task to force us to eventually get to c.

m1-goto(*dest*)	
task:	goto(*dest*)
assertions:	$[t_{start}, t_{end}]\, at = dest$
subtasks:	$\varnothing$
constraints:	$\varnothing$

m2-goto($from$, d', $dest$)	
task:	goto($dest$)
assertions:	$[t_{start}]\, at = from$
subtasks:	$[t_{start}, t_1]\, move(from, d')$
	$[t_2, t_{end}]\, goto(dest)$
constraints:	$from \neq dest \wedge t_1 < t_2$

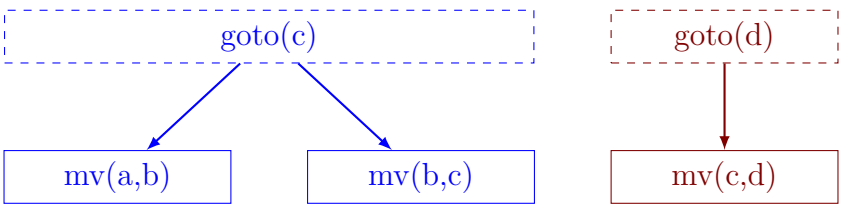
This problem can be avoided by making sure the same original task is not used to get to the same place twice. This can be achieved by adding to the second action the set of assertions:

$$\begin{aligned} & [t_{start}, t_{start}+1]\, visited(from) : \bot \mapsto \top \\ \wedge\ & [t_{start}+1, t_{end}-1]\, visited(from) = \top \\ \wedge\ & [t_{end}-1, t_{end}]\, visited(from) : \top \mapsto \bot \end{aligned}$$

These statements make sure that a place p is not visited by a second action as it would force changing the value of the $visited(p)$ state variable while it is protected by the first action. This essentially attaches a unary resource to each location and makes sure it is not used twice. As a result two instances of `m2-goto`$(\mathtt{x}, \cdot)$ are mutually exclusive (independently of their second parameter).

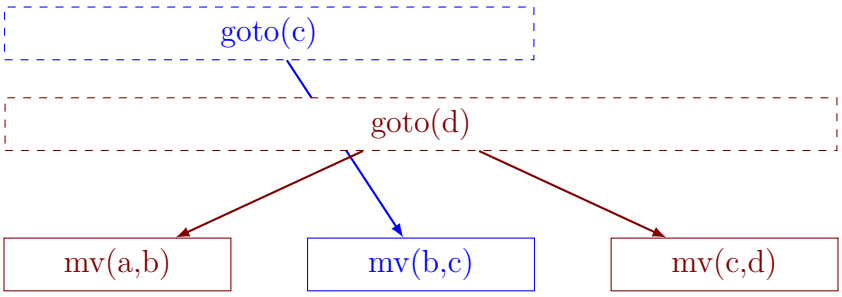
This could even be expressed in a single (less intuitive) change assertion $\langle [t_{start}, t_{end}]\, visited(from) : \bot \mapsto \bot \rangle$. Indeed it would force two durative changes on the same state variable to overlap.

Another trap for a hierarchical planner is the presence of redundancies in the search space. Indeed, a hierarchical planner usually considers a plan as many times as there are ways to decompose it from the initial task network, leading to redundancies in the search space. For this reason, the domain designer must make sure that there is indeed only one way of decomposing it. Consider for instance that, being in a location a, we want to go to a location c and to a location d without any particular order. Expressed with two tasks $goto(c)$ and $goto(d)$, we probably intend the planner to find a solution like this one:



where the $goto(c)$ task contributes the first two actions to get me from a to c (in blue) and the $goto(d)$ task contributes a last action to get me from c to d (in red).

However we certainly didn't expect the planner to come up with this equivalent solution where the first move is contributed by the *goto*(*d*) task:



However, it is a perfectly valid solution, equivalent to the first one regarding the sequence of movements. But it lies in a different branch of the search space because the planner took two different paths to decompose it. One possible way to avoid such issues is to make actions whose interactions lead to non-systematicity mutually exclusive with a similar trick as the one used in Example 3.6. Indeed, forbidding the two highest-level *goto* tasks to overlap would maintain completeness while enforcing systematicity. This should however be done with care as many interactions can be beneficial and forbidding those can easily lead to a loss of completeness.

As we can see, writing fully hierarchical domains requires care and practice. The fact that FAPE is able to solve most of the problems it faces in a generative fashion or with partial hierarchies is enjoyable. Nevertheless, the ability to do HTN planning can be the key to solve problems that would have been inaccessible to the domain-independent setting, as best seen for the *hiking* and *turnandopen* domains in our tests set.

In other cases, some problems are more naturally expressed in a hierarchical fashion that maps to a procedural knowledge of the domain designer. This is the case of the *race* domain, where constraints on the navigation of the robots appear as procedural knowledge in high-level actions (Section B.3).

3.5.3 Comparison with CHIMP

Because of the important similarities of representation but large difference in search techniques, we empirically compared the performance of both FAPE and CHIMP on two domains. The first one was defined in the scope of the European project RACE for use in CHIMP and has been translated in a fully hierarchical ANML domain. The RACE domain deals with service robots that must serve clients by bringing the order, while obeying complex navigation constraints. The second domain is the dock worker that we have used as an example through this dissertation and that has been translated by CHIMP's authors for use in their planner. For both domains, we ensured that both planners have equivalent models: each one has the same primitive and high-level actions with similar conditions and effects. We compare the planners on several planning problems that involve one or multiple planning tasks: respectively of bringing a container to certain location or serving a customer. In each case achieving a single task requires between 12 and 16 primitive actions that are either sequenced or independent (i.e. CHIMP was able to impose an ordering constraint on their decomposition). The runtime for solving problems in those two domains is presented in Table 3.8.

CHIMP was able to solve all problems with a single goal task, regularly outperforming

Dock Worker Domain

Number of tasks	1		2			3		4	
FAPE	1.17	0.89	1.88	3.03	0.94	3.44	12.43	27.34	–
CHIMP	29.18	0.76	203.26	0.71	–	2.31	–	–	–

RACE Domain

Number of tasks	1		2				3	4	5
FAPE	1.29	0.77	1.27	1.42	1.22	1.25	1.91	2.74	4.13
CHIMP	0.70	0.63	–	–	5.12	–	–	–	–

Table 3.8: Runtime (in seconds) for solving hierarchical planning problems depending on the number of goal tasks. A field with – indicates that the planner was not able to solve the task within 10 minutes. Both domains and the problem instances are available in FAPE's public repository.

FAPE on those.[3] The catch with CHIMP occurs as soon as more than two tasks are involved. Indeed in such cases, imposing an order on the decomposition of the goal tasks is not possible because they are not independent. In this case, if it does not find a solution on the first try, CHIMP must eventually backtrack on the choice of the order in which tasks are decomposed, leading to much redundancy in the search space. As a result, CHIMP usually finds a solution quickly or times out. On the other hand, the more involved approach of FAPE allows it to better scale to planning problems with multiple tasks. It is however important to note that this result is not final: since CHIMP mostly relies on domain-dependent knowledge, its scalability could most certainly be improved by extending the domain definition. This experiment simply highlights the advantage that FAPE has over CHIMP when the two planners are given the same hierarchical knowledge. On the other hand, since it is based on the meta-CSP framework, CHIMP can directly benefit from the extensions already built in the framework, e.g., CHIMP supports reasoning on resources which FAPE currently does not.[4]

3.6 Conclusion

In this chapter, we have presented a planning algorithm for FAPE that allows a unique mix of generative and hierarchical temporal planning. The planner reasons in plan-space with a least-commitment approach and integrates several inference techniques to restrict its search space. The planner uses a new reachability analysis that adapts the delete-free relaxation to a temporal and hierarchical setting. Further techniques were developed to extract constraints and heuristics from the causal structure of a partial plan. We have shown the planner to be competitive with state of the art domain-independent temporal planners when used in a generative setting. Its additional support for hierarchies allows to further extend both its efficiency and its expressivity.

While the planner we have presented focuses on causal reasoning, its constraint based

[3]We also noticed this behavior on many test instances with simpler goal tasks that are not presented here.

[4]Partial support for resources has been implemented in FAPE but is still regarded as unstable and is not well integrated with the other aspects of the search.

representation has several advantages when its comes to considering real-world problems. First, by extending its flaw detection functions to reason on other particular subproblems, the planner can naturally be extended to handle richer problems than most AI planning systems. Timeline based planners have a strong history in developing such extensions (see Section 3.4.3) most of which could be integrated in FAPE. The next chapter will demonstrate how such an extension is possible by augmenting the planer to reason under temporal uncertainty. Second, the plan-space approach has some key advantages when it comes to continual planning and acting, as once again demonstrated by many timeline based planners. The last chapter of this thesis will be dedicated to show how FAPE takes advantage of this in the context of deliberative acting for robotics.

Chapter 4

Planning under Temporal Uncertainty

Contents

4.1 Introduction

So far, we have only considered problems in which all temporal aspects are either exactly known or under the control of the planning system. This is however a limiting assumption when considering real-world problems. For instance, the duration of a trip from Toulouse to Paris is always reliably predictable in a limited interval. However, its actual duration is not entirely under the control of the driver but will instead depend on the weather and the traffic. In the context of task planning, such contingencies typically appear on the duration of actions (e.g. duration of a trip) or in the definition of exogenous events (e.g. the time at which a guest arrives).

Handling such contingencies can be done in several ways. A planner could simply disregard any uncertainty in the duration and instead use a fixed estimation of the duration. One could also try to build temporally flexible plans that make a best-effort not to depend on any precise duration of contingent events. We follow a more involved approach which is to guarantee a plan to be executable regardless of the outcome of contingent durations.

In this chapter, we explicitly model the temporal uncertainty by considering contingent events that are not under the control of the actor. Contingent events are described by explicit lower and upper bounds on their occurrence time relatively to another event. Given a problem with contingent events, the objective of the planner is to find a plan that is valid regardless of the occurrence time of contingent events.

We start by introducing the related work around the Simple Temporal Problem with Uncertainty (STPU) that will serve as a base for modeling temporal uncertainty. We then analyze some of the limitations of the STPU, especially for representing problems where only part of the environment is observable. We lift those limitations in a generalization of the STPU that incorporates the notion of partial observability and propose dedicated algorithms to check whether a plan based on our extended model is executable. Last, we show how the framework is integrated in FAPE and provide techniques to reason on which observations are needed to maintain a plan dispatchable. We show how such needed observation can be identified at planning time and incrementally dealt with by considering the appropriate sensing actions. We conclude on some empirical evaluation of the proposed techniques and some discussions.

4.2 Background and Related Work

4.2.1 Background

4.2.1.1 Simple Temporal Network with Uncertainty

A Simple Temporal Network with Uncertainty (STNU) extends an STN with *contingent links*, where each contingent link represents a temporal interval whose duration is not under the control of the agent. Contingent links are typically used to represent uncertain durations of actions (e.g. the duration of the trip in our previous example) and uncertainty on the occurrence time of exogenous events (e.g. arrival time of a guest).

Definition 4.2.1 (STNU)**.** An STNU is a triple $(\mathcal{X}, \mathcal{R}, \mathcal{C})$ where $(\mathcal{X}, \mathcal{R})$ is an STN composed of a set of timepoints $\mathcal{X}$ and a set of requirement link $\mathcal{R}$ and $\mathcal{C}$ is a set of contingent links:

- As in an STN, a *requirement* link $C \xrightarrow{[y,y']} D$ states that the delay between C and D should be in the interval $[y, y']$.
- A *contingent* link $A \xRightarrow{[x,x']} B$ states that the delay between the events A and B is non-controllable and will be fixed by the environment in the interval $[x, x']$.

This definition naturally leads to two types of timepoints: (i) *contingent events* have incoming contingent constraints and their occurrence time is fixed by the environment (within the bounds specified on their incoming contingent link); (ii) *controllable events* are events that have no incoming contingent links and are to be dispatched (i.e. assigned

an execution time) by the execution process. We denote the set of contingent points by $\mathcal{X}_{\text{ctg}}$ and the set of controllable events by $\mathcal{X}_{\text{ctl}}$, where $\mathcal{X} = \mathcal{X}_{\text{ctl}} \cup \mathcal{X}_{\text{ctg}}$ and $\mathcal{X}_{\text{ctl}} \cap \mathcal{X}_{\text{ctg}} = \varnothing$.

Simple Temporal Networks with Uncertainty were first introduced by Vidal and Ghallab [VG96], and refined by Vidal and Fargier [VF99], to handle the presence of uncertain durations in planning. A typical problem with an STNU is a plan where the duration of actions is uncontrollable. In such a case, the start time of actions are denoted by controllable timepoints and the plan is controllable if one can devise a viable strategy for deciding when each action is to be dispatched (while ensuring that no temporal constraints are violated). Such a strategy can be either static (the dispatch times are precomputed before starting the execution of the plan) or dynamic (the values of controllable timepoints are assigned online, allowing to take into account knowledge already accumulated during the execution). Vidal and Fargier [VF99] propose three ways of characterizing the controllability of an STNU depending on the assumption that can be made regarding the time at which the duration taken by a contingent will be known. Namely, they introduce the notion of *strong controllability*, *weak controllability* and *dynamic controllability*. Before describing those, we give some definitions that will be useful in characterizing them.

An STNU can represent several situations, each associating all contingent links with the duration they will take.

Definition 4.2.2 (Situations)**.** If $A_1 \xrightarrow{[l_1,u_1]} B_1, \ldots, A_K \xrightarrow{[l_K,u_K]} B_K$ are the K contingent link in an STNU $\mathcal{S}$, we call $\Psi_{\mathcal{S}}$ the space of situations of $\mathcal{S}$, given by $\Psi_{\mathcal{S}} = [l_1, u_1] \times \cdots \times [l_K, u_K]$.

Each situation $\psi = (\psi_1, \ldots, \psi_K) \in \Psi_{\mathcal{S}}$ gives one possible value for each of the contingent links in $\mathcal{S}$.

A situation essentially maps each contingent link to the value it will take. For a given situation $\psi \in \Psi_{\mathcal{S}}$, there is no uncertainty regarding the contingent links and the STNU $\mathcal{S}$ can thus be translated to an STN $\mathcal{S}_\omega$.

Definition 4.2.3 (Projection)**.** A projection of the STNU $\mathcal{S} = (\mathcal{X}, \mathcal{R}, \mathcal{C})$ onto the situation ψ is defined to be the STN $(\mathcal{X}, \mathcal{R} \cup \mathcal{C}_\psi)$ where $\mathcal{C}_\psi$ is the set of links where each contingent constraint $A_i \xrightarrow{[l_i,u_i]} B_i \in \mathcal{C}$ is replaced by a requirement link $A_i \xrightarrow{[\psi_i,\psi_i]} B_i$ (i.e. $\mathcal{C}_\psi = \{\, A_i \xrightarrow{[\psi_i,\psi_i]} B_i \mid 1 \leq i \leq K \,\}$).

For a given STNU, we are typically interested in finding a schedule that gives a dispatch time to each event under our control.

Definition 4.2.4 (Schedule)**.** A schedule ξ of an STNU $\mathcal{S} = (\mathcal{X}_{\text{ctl}} \cup \mathcal{X}_{\text{ctg}}, \mathcal{R}, \mathcal{C})$ is a mapping $\xi : \mathcal{X}_{\text{ctl}} \to \mathbb{Z}$ from controllable timepoints to integer values.[1]

A schedule thus maps every controllable timepoints to a time at which it is executed. We note that knowing the situation and a schedule is sufficient to know the occurrence time of all (contingent and controllable) timepoints. Given a schedule ξ and a situation ψ, we denote as $[x]_{\xi,\psi}$ the time at which the timepoint $x \in \mathcal{X}$ would be executed.

Furthermore, we say that a schedule $\xi : \mathcal{X}_{\text{ctl}} \to \mathbb{Z}$ is consistent with an STN $(\mathcal{X}, \mathcal{R})$, with $\mathcal{X}_{\text{ctl}} \subseteq \mathcal{X}$, iff $(\mathcal{X}, \mathcal{R} \cup \mathcal{R}^{sched})$ is a consistent STN where $\mathcal{R}^{sched}$ encode the occurrence times imposed by the schedule:

[1] In this definition, the execution of time of events is limited to the set of integers since that fits our temporal model. In general though, STNUs are not restricted to discrete time.

$$\mathcal{R}^{sched} = \bigcup_{x \in \mathcal{X}_{\mathfrak{ctl}}} \mathcal{O} \xrightarrow{[\xi(x),\xi(x)]} x$$

4.2.1.2 Strong Controllability

A first interesting property for an STNU is whether it is *strongly controllable*, i.e., whether there exist a single schedule that will work for all situations.

Definition 4.2.5 (Strong Controllability)**.** An STNU $\mathcal{S}$ is *strongly controllable* if there is a schedule ξ such that for all situations $\psi \in \Psi_{\mathcal{S}}$, ξ is consistent with $\mathcal{S}_\psi$.

Essentially, strong controllability denotes the fact that an STNU is controllable (i.e. the corresponding plan can be executed) when the agent can schedule a priori all controllable points without knowledge on any contingent events beside the bounds initially provided.

Strong controllability can be detected in polynomial time, by compiling it into an equivalent STN [VF99]. The key to this compilation is to transform each constraint c involving a contingent timepoint $x \in \mathcal{X}_{\mathfrak{ctg}}$ into two constraints c_{max} and c_{min} where c_{max} (resp. c_{min}) enforces that c is respected in the eventuality that x occurs at its latest (resp. at its earliest).

4.2.1.3 Weak Controllability

The complementary of strong controllability is *weak controllability*, i.e., that, if the situation is known before execution, then one can find a schedule that is valid for this situation.

Definition 4.2.6 (Weak Controllability)**.** An STNU $\mathcal{S}$ is *weakly controllable* if for all situations $\psi \in \Psi_{\mathcal{S}}$, there is a schedule ξ_ψ such that ξ_ψ is consistent with $\mathcal{S}_\psi$.

Essentially, weak controllability denotes the fact that an STNU is controllable when the agent has complete knowledge on all contingent events before starting the execution of a plan. This is equivalent to saying that every projection of an STNU is a consistent STN.

Weak controllability has been shown to be co-NP-complete by Vidal and Fargier [VF99]. The same authors also provide an algorithm for it that essentially checks if all relevant projections are consistent, where a projection is relevant when all contingent durations take either their minimum or maximum values.

4.2.1.4 Dynamic Controllability

Dynamic controllability (DC) is a last characterization of an STNU where an agent is interested in having a dynamic execution strategy where the dispatch time of actions is decided online and can leverage accumulated knowledge regarding the occurrence of past events. A network is *dynamically controllable* if the dispatch time of controllable events can be decided online assuming a knowledge on past events only.

We first define an execution strategy as a mapping from a given situation to a schedule and say that it is viable if it does not result in any violated constraint. We will then refine this concept to a dynamic execution strategy where the situation is not given but partially characterized by the observation of past events.

Definition 4.2.7 (Execution Strategy)**.** An execution strategy S for an STNU $\mathcal{S}$ is mapping $S : \Psi_{\mathcal{S}} \to \Xi$ from situations to schedules (where Ξ denotes the set of possible schedules).

Definition 4.2.8 (Viable Execution Strategy)**.** An execution strategy S, is said *viable* for an STNU $\mathcal{S}$ if for every situation $\psi \in \Psi_{\mathcal{S}}$, $S(\psi)$ is a schedule consistent with $\mathcal{S}_\psi$, the projection of $\mathcal{S}$ onto ψ.

In the case of dynamic controllability however, the situation is not entirely known but instead, the agent only knows the part that correspond to past events, which we call the execution trace.

Definition 4.2.9 (Execution trace)**.** The execution trace up to time t of applying a schedule ξ to a situation ψ, is defined by the occurrence time of all contingents timepoints that occurred before t. This execution trace is denoted by $\Theta^t_{\xi,\psi}$:

$$\Theta^t_{\xi,\psi} = \{\, (x, [x]_{\xi,\psi}) \mid x \in \mathcal{X}_{\text{ctg}} \text{ and } [x]_{\xi,\psi} < t \,\}$$

An execution strategy is *dynamic* if the decision of dispatching a timepoint at a given time t only depends on the (past) execution trace.

Definition 4.2.10 (Dynamic Execution Strategy)**.** A strategy S is said to be a *dynamic execution strategy* if for any two situations ψ and ψ', and any timepoint $x \in \mathcal{X}_{\text{ctl}}$, the following holds:

$$\text{if } \xi_\psi(x) = t \text{ and } \Theta^t_{\xi_\psi,\psi} = \Theta^t_{\xi_{\psi'},\psi'} \text{ then } \xi_{\psi'}(x) = t$$

where ξ_ψ is the schedule selected by S for the situation ψ (i.e. $S(\psi)$).

Definition 4.2.11 (Dynamic Controllability)**.** An STNU is dynamically controllable if it has a viable dynamic execution strategy.

Example 4.1. Consider the following problem where an agent has to prepare dinner for its partner and wants to ensure that the meal is ready (or almost ready) and still warm when its partner gets home. The agent has knowledge about the planned activities of its partner: he will be we working for 30 to 60 minutes then drive home which will take between 35 and 40 minutes. He further knows that preparing dinner will take from 25 to 30 minutes. With all this information, the agent should decide when to start cooking so that the dinner be ready when his partner gets home (with a five minutes margin before and after this moment).

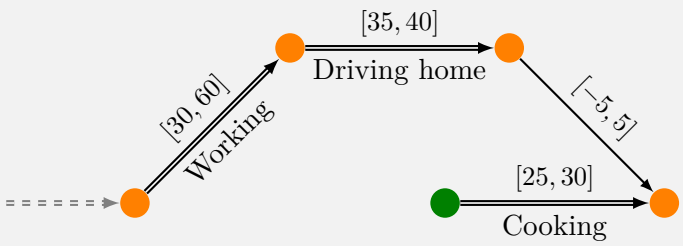


Figure 4.1: A dynamically controllable STNU, with contingent timepoints (in orange) representing the uncontrollable activities of a partner. A controllable timepoint (in green) denotes the moment when one should start cooking so that the dinner is ready at most 5 minutes before or after one's partner gets home.

The solution to this problem is to start cooking exactly 10 minutes after the partner leaves work. Indeed, this strategy means that the events of the dinner being ready and of our partner arriving home will both occur within $[t_{drive} + 35, t_{drive} + 40]$, where t_{drive} denotes the moment our partner starts driving. This strategy is guaranteed to meet all constraints (and is in fact the only one that does).

The strategy simply requires to know the occurrence time of t_{drive} in order to dispatch the t_{cook} timepoints at $t_{drive} + 10$. The network is thus dynamically controllable since dispatching the plan requires knowledge on past events only.

A characterization of Dynamic Controllability was first proposed by Vidal and Fargier [VF99] and more formally exposed by Morris, Muscettola, and Vidal [MMV01]. Hunsberger [Hun09] later amended this definition to disallow the instantaneous reaction to contingent events. It is his definition that we presented here, with some minor changes to ease the joint presentation of weak and strong controllability. Hunsberger [Hun09] further proposed a semantically equivalent definition centered on the real-time execution decision that a planning agent must make.

The first technique for checking dynamic controllability relied on the translation to a *timed game automata* [VF99]. This untractable method was abandoned for constraint propagation techniques that were initially pseudo-polynomial [MMV01]. Polynomial versions have been proposed by Morris and Muscettola [MM05] (with a runtime quadric in the number of timepoints) that were refined by Morris [Mor14] and Hunsberger [Hun14] to run with a cubic complexity.

An incremental algorithm was introduced by Shah et al. [Sha+07] and also relies constraint propagation. This algorithm was corrected and improved by Nilsson, Kvarnström, and Doherty, whose latest algorithm also has a cubic complexity [NKD13; NKD14b; NKD15].

Recent algorithms for DC-checking rely on a *labeled distance graph* representation that is convenient for encoding propagation rules. It relies on transforming the STNU network into a labeled multigraph obtained as follows. Each requirement link $A \xrightarrow{[x,y]} B$ is replaced by two edges $A \xrightarrow{y} B$ and $A \xleftarrow{-x} B$. A contingent link $A \xRightarrow{[x,y]} B$ is replaced by four edges: the same two edges as for a controllable link, and two labelled edges $A \xrightarrow{b:x} B$ and $A \xleftarrow{B:-y} B$, called *lower-case* and *upper-case* edges, referring respectively to the lower and upper bound values of the contingent link.

A labeled distance graph can be used to compute distances between nodes, as in a distance graph of an ordinary STN. An STN is consistent if and only if its distance graph does not contain a negative cycle. Specific propagation rules (Table 4.1) have been devised for labelled distance edges to provide a similar property: an STNU is dynamically controllable iff it does not have a so-called *semi-reducible negative cycle* obtained with these constraint propagation rules [Mor14]. A procedure DC-Check takes as input an STNU; it propagates its constraints with a Dijkstra-like algorithm; it returns *true* when no such a negative cycle is found, which entails a DC STNU.

4.2.2 Other Related Work

Beyond the work regarding the characterization of an STNU that we discussed above, some work has been dedicated to providing algorithms for efficient execution of dynamically controllable networks [Hun10; Hun13; Mor14; Hun16].

Conditions	Added constraint
$A \xleftarrow{B:x} C \xleftarrow{y} D$	$A \xleftarrow{B:(x+y)} D$
$A \xleftarrow{x} C \xleftarrow{c:y} D, \quad x < 0$	$A \xleftarrow{x+y} D$
$A \xleftarrow{B:x} C \xleftarrow{c:y} D, \quad x < 0, B \neq C$	$A \xleftarrow{B:(x+y)} D$
$A \xleftarrow{x} C \xleftarrow{y} D$	$A \xleftarrow{x+y} D$
$B \xleftarrow{b:x} A \xleftarrow{B:z} C, \quad z \geq -x$	$A \xleftarrow{z} C$

Table 4.1: Constraint propagation rules in an STNU

The encoding of a timeline-based plan with uncontrollable durations into a Timed Game Automata (TGA) has been recently used by Mayer and Orlandini [MO15] to check its controllability. This approach has the advantage of relying on well established model checking tools to assess the controllability of the STNU and synthesize an execution strategy that can be efficiently executed online. Its main disadvantage when compared to recent propagation based techniques to dynamic controllability is its large overhead that limits its usage to an offline setting.

Some extensions to the STNU framework have been considered in the literature. The Disjunctive Temporal Network with Uncertainty (DTNU) allows the presence of disjunctive constraints. Algorithms have been proposed to check the strong controllability [CMR12a; CMR14] and the weak controllability [CMR12b; CMR15] of a DTNU that rely on a translation to SMT. More specifically, quantitative constraints are encoded into the theory Linear Arithmetic over The Real Numbers with additional boolean variables encoding the disjunctive parts of the problem. The same authors have further studied the dynamic controllability of a DTNU by mean of a translation to a Timed Game Automata [Cim+14a; Cim+14b; Cim+16].

Combi, Hunsberger, and Posenato [CHP14] have considered Conditional STNU (CSTNU) where the actions taken by an agent are conditioned regarding the occurrence of contingent events. A translation of Conditional and Disjunctive STNU into TGAs was given by Cimatti et al. [Cim+16]. Lanz et al. [Lan+15] consider a special case of CSTNUs were an uncertain duration can be reduced in order to meet a deadline (e.g. allowing the agent to perform a strictly shorter activity instead).

Moffitt [Mof07] is the first to consider the case where the occurrence of only a subset of the events can be dynamically observed and proposes an algorithm to characterize the dynamic controllability of such networks through updated propagation rules. Their algorithm is unsound and can only determine when the network is not dynamically controllable. Casanova et al. [Cas+16] study a similar case where multiple agents must carry out a shared plan where an agent can only know the duration taken by its own actions. They check the dynamic controllability of such network by a sound but not complete translation to MILP.

4.3 Generalized Controllability of STNU

So far, we have presented ways of characterizing an STNU as:

- *strongly controllable*, meaning that the network is dispatchable without any requirement on the observation of contingent events,

- *dynamically controllable*, meaning that the network is dispatchable given that all contingent events are observed when they occur,
- *weakly controllable*, meaning that the network is dispatchable given that the duration of all contingent links is known before starting the execution.

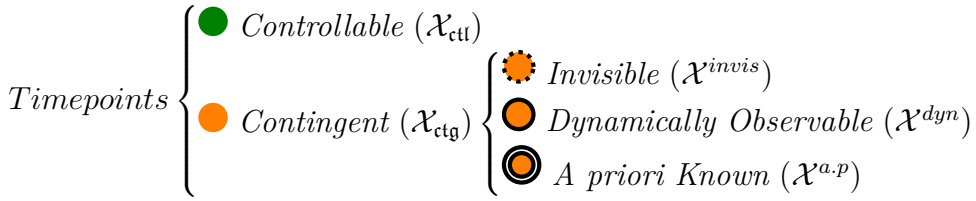
Such a view is however limiting in many cases. Consider our cooking example of Figure 4.1. While I can certainly observe the events corresponding to my partner arriving home and to me finishing cooking, its is unlikely that I know the moments when he starts working or starts driving home. The case where an STNU contains observable contingent events and non-observable ones is not captured by neither dynamic controllability (that assumes total observation of all contingent events) nor strong controllability (that does not account for any observations). In fact in this case, different timepoints of the network depend upon different types of controllability.

We can find scenarios where weak controllability is relevant. Indeed, task planning often serves as a means to produce an high-level plan for which many details have not yet been planned, either because they are too expensive to compute or because some information is lacking. Consider for instance an action involving a robot picking an object. Its duration is at first unknown to the task planner. However, once a complete motion plan has been computed for this action its duration will be known. Such a case where some contingent durations are known prior to execution comes within the competency of weak controllability. Here again, it is incorrect to assume that all other contingent events are in this same category.

To model such cases, we define a Partially Observable STNU (POSTNU) as an extension to an STNU where the observability of contingent events is explicit.

Definition 4.3.1 (POSTNU)**.** We define a POSTNU as a tuple $\Omega = (\mathcal{X}_{\mathfrak{ctl}}, \mathcal{X}_{\mathfrak{ctg}}, \mathcal{R}, \mathcal{C})$ where the set of contingent points $\mathcal{X}_{\mathfrak{ctg}}$ is partitioned into:

- a set of *a priori known* events $\mathcal{X}^{a.p}$, whose occurrence time is currently unknown but will be known prior to execution (relatively to another event),
- a set of *dynamically observable* events $\mathcal{X}^{dyn}$, whose occurrence time will be known when they occur,
- a set of *invisible* events $\mathcal{X}^{invis}$ whose occurrence cannot be observed.



We say that a POSTNU is controllable if there is a viable execution strategy that depends on the value taken by *a priori known* events and past *dynamically observable* events only.

Definition 4.3.2 (Visible Execution Trace)**.** We call the *visible execution trace* of up to time t of applying a schedule ξ to a situation ψ, the occurrence time of all dynamically

observable events that occurred before t and the occurrence time of all *a priori known* events. This execution trace is denoted by $\Gamma^t_{\xi,\psi}$:

$$\Gamma^t_{\xi,\psi} = \{\, (x, [x]_{\xi,\psi}) \mid x \in \mathcal{X}^{a.p} \vee x \in \mathcal{X}^{dyn} \wedge [x]_{\xi,\psi} < t \,\}$$

An execution strategy S is dynamic if the choice of executing a controllable timepoint at time t depends only on contingent events that have been observed at time t: either *a priori known* events whose value was known before execution or *dynamically observable* events that occurred prior to time t.

Definition 4.3.3 (Dynamic Execution Strategy of a POSTNU)**.** A strategy S is said to be a *dynamic execution strategy* if for any two situations ψ and ψ', and any timepoint $x \in \mathcal{X}_{\text{ctl}}$, the following holds:

$$\text{if } \xi_\psi(x) = t \text{ and } \Gamma^t_{\xi_\psi,\psi} = \Gamma^t_{\xi_{\psi'},\psi'} \text{ then } \xi_{\psi'}(x) = t$$

where ξ_ψ is the schedule selected by S for the situation ψ (i.e. $S(\psi)$).

Definition 4.3.4 (Controllability of a POSTNU)**.** A POSNTU is said to be *controllable* if it has a viable dynamic execution strategy.

This definition of POSTNU relates to strong, weak and dynamic controllability in the special cases where all contingent events are all *invisible*, *a priori known* or *dynamically observable* respectively.

Example 4.2. Below is a revised version of our cooking example where only some of the contingent events are dynamically observable.

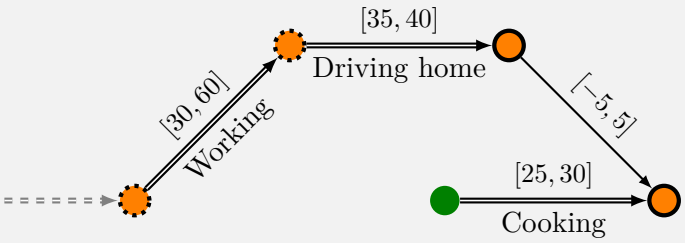


Figure 4.2: A revised version of Figure 4.1 where the only dynamically observable events are the ones of our partner arriving home and of us finishing cooking.

4.4 Checking the controllability of a POSTNU

In this section we are interested in devising techniques to check whether a given POSTNU is controllable. Our main interest lies in finding tractable algorithms that can be used at planning time while giving a guaranty on the controllability of the plan.

A first obstacle to this objective is the presence of *a priori known* timepoints. Indeed, checking the weak controllability of an STNU (i.e. the controllability of a POSTNU where all contingent timepoints are *a priori known*) has been shown to be co-NP-complete [VF99]. A simple trick to avoid this difficulty is to consider every contingent link $A \overset{[l,u]}{\Longrightarrow} B$,

where B is *a priori known*, as a requirement link $A \xrightarrow{[l,u]} B$. One can then ensure that the original link is not *squeezed* during propagation, i.e., that no new constraint $A \xrightarrow{[l',u']} B$ is inferred where $l < l'$ or $u' < u$. While this is a necessary condition of the controllability of the POSTNU it is not a sufficient one but can serve as a tractable relaxation [MMV01].

Another tractable approach is to consider all *a priori known* timepoints as *dynamically observable*. The resulting POSTNU assumes strictly less information about the environment, thus proving its controllability is sufficient to demonstrate the controllability of the original network. The converse is however not true.

For the purpose of finding tractable algorithms, we restrict ourselves to the case where the POSTNUs contains no *a priori known* events. Such events could have been compiled away by one of the two methods above depending on whether they should be considered optimistically or pessimistically.

In order to check the controllability of POSTNU, we map it into an STNU. The key idea is to locally apply the propagation rules of Strong Controllability to invisible events. Once those are propagated, the invisible points, and the constraints involving them, can be removed from the network. The resulting network contains only *dynamically observable* points and is controllable if the corresponding STNU is dynamically controllable.

This is performed by procedure PO-Contr-Check (Algorithm 3). For every invisible node $x^{inv} \in \mathcal{X}^{invis}$, it propagates the constraints in which it appears using the propagation rules of Table 4.2 (PropagateInvisible). All edges to and from x^{inv} are then removed from the network (RemoveEdges). At that point $(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{dyn}, \mathcal{R}', \mathcal{C}')$ is an STNU whose dynamic controllability is tested with DC-Check.

Algorithm 3 Maps a POSTNU into a dynamically observable STNU whose controllability is tested with DC-Check.

function PO-Contr-Check($\mathcal{X}_{\text{ctl}}, \mathcal{X}^{dyn}, \mathcal{X}^{invis}, \mathcal{R}, \mathcal{C}$)
  $(\mathcal{R}', \mathcal{C}') \leftarrow (\mathcal{R}, \mathcal{C})$
  for all $x^{inv} \in \mathcal{X}^{invis}$ **do**
    $(\mathcal{R}', \mathcal{C}') \leftarrow$ PropagateInvisible$(\mathcal{R}', \mathcal{C}', x^{inv})$
    $(\mathcal{R}', \mathcal{C}') \leftarrow$ RemoveEdgesInvolving$(\mathcal{R}', \mathcal{C}', x^{inv})$
  return DC-Check$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{dyn}, \mathcal{R}', \mathcal{C}')$

Conditions	Added constraint
$A \xleftarrow{B:x} B \xleftarrow{y} C$	$A \xleftarrow{x+y} C$
$A \xleftarrow{B:x} B \xleftarrow{C:y} C$	$A \xleftarrow{C:x+y} C$
$A \xleftarrow{x} B \xleftarrow{b:y} C$	$A \xleftarrow{x+y} C$
$A \xleftarrow{a:x} B \xleftarrow{b:y} C$	$A \xleftarrow{a:x+y} C$

Table 4.2: Added constraints due to an invisible point B.

Proposition 4.4.1. *If* PO-Contr-Check$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{dyn}, \mathcal{X}^{invis}, \mathcal{R}, \mathcal{C})$ *returns true, then the POSTNU* $\mathcal{S} = (\mathcal{X}_{\text{ctl}}, \mathcal{X}^{dyn} \cup \mathcal{X}^{invis}, \mathcal{R}, \mathcal{C})$ *is controllable.*

Proof. Consider an invisible timepoint B with an incoming contingent link $A \xRightarrow{[l,u]} B$ (i.e., $A \xrightarrow{b:l} B$ and $A \xleftarrow{B:-u} B$ in the labeled distance graph). B cannot be the target of

more than one contingent link, otherwise the network is not DC. There are three possible constraints that may involve B:

- B is the target of a requirement edge $B \xleftarrow{-y} C$, which says *"C must occur at least y time units after B"*. The first rule in Table 4.2 adds the edge $C \xleftarrow{-y-u} A$. This edge dominates the original one: a minimum delay of at least y will be maintained between B and C regardless of the outcome of the contingent link $A \Rightarrow B$.

- B is the source of a requirement edge $B \xrightarrow{x} C$, which says *"C should occur at most x time units after B"*. The third rule in Table 4.2 introduces an edge $A \xrightarrow{x+l} C$. The original edge is dominated by this edge: a maximum delay of x is enforced between B and C regardless of the outcome of the contingent link $A \Rightarrow B$.

- B is the source of a contingent link $B \overset{[x,y]}{\Longrightarrow} C$, giving two edges $B \xrightarrow{c:x} C$ and $B \xleftarrow{C:-y} C$. The second and fourth rules in Table 4.2 add the edges $A \xrightarrow{c:x+l} C$ and $A \xleftarrow{C:-y-l} C$. These merge the two contingent links $A \Rightarrow B \Rightarrow C$ into one, ignoring B as an intermediary point. They dominate the original ones as they are less informative.

Consequently, any edge involving B in the original POSTNU is dominated by an edge introduced by the rules in Table 4.2. All those edges to and from B can thus be removed from the STNU without relaxing the problem. □

The converse of Proposition 4.4.1 does not hold in general, since the transformation of PO-Contr-Check is conservative on some POSTNUs. An example of such network is given in Figure 4.3. This network has a *chained contingency*: there is a contingent timepoint B that is at the center of a contingent chain $A \Rightarrow B \Rightarrow C$ and is involved in an additional contingent or requirement link with another timepoint D.

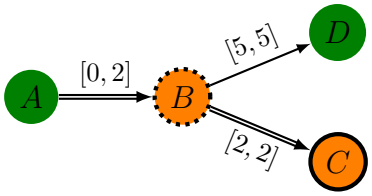


Figure 4.3: A POSTNU where PO-Contr-Check is too conservative. The network is controllable even if B is invisible: just schedule D 3 time units after C. Here, B is *indirectly observable* through C.

The converse of Proposition 4.4.1 holds for a POSTNU without chained contingencies.

Proposition 4.4.2. *A POSTNU $\mathcal{S}$ with no chained contingencies is controllable if and only if* PO-Contr-Check($\mathcal{S}$) *holds.*

Proof. It is sufficient to show that, for a given contingent point B, either *(i)* no indirect information can be gathered on B after its occurrence; or *(ii)* such information is useless since B is not involved in any requirement link. □

POSTNU without chained contingencies are in general relevant in a planning context: contingent points in a plan are usually the end points or the intermediate points of the agent own actions as well as the expected events to be triggered by the environment

independently of its actions. The expectations regarding the latter are often known with respect to absolute time (e.g. periodic events), which rule out chained contingencies. Expected events, supposed to occur regardless of the agent activity, can seldom be defined with respect to the end points of its actions. Further, two consecutive actions cannot make a chained contingency. Remains as a case of such contingencies intermediate and final points of actions. This case can be avoided by decomposing the action or expressing the contingent constraints with respect to the controllable point. If this is not convenient, then PO-Contr-Check loses completeness but remains sound and useful in practice.

The complexity of PO-Contr-Check is in $O(|\mathcal{X}|^3)$ for a network with the set of timepoints $\mathcal{X}$: the *for* iteration is in $O(|\mathcal{X}^{invis}| \times |\mathcal{R} \cup \mathcal{C}|)$ dominated by the DC-Check step of cubic complexity when using current state of the art algorithm [Mor14].

Applied on our cooking example of Figure 4.2, PO-Contr-Check would show that this POSTNU is uncontrollable.

4.5 Finding what to observe in a POSTNU

We partitioned contingent events into *invisible* and *dynamically observable* ones. However, an observable event may or may not be visible to an agent depending on, e.g., its location, the setting of its sensors or its concurrent activity. The agent may need to perform specific actions to perceive observable events.

In our cooking example, it would be interesting to know if additional knowledge would make the problem controllable. In this case, one could infer that knowing when his partner leaves work will allow him to start cooking 10 minutes after. Such additional information can easily be requested to the other agent (e.g. with a "text me when you leave" request). This request essentially turns an invisible event into an observable one.

As a first step to achieve this deliberation, we study how an agent can find which observations are necessary to make an uncontrollable POSTNU controllable.

Given a plan π, we further partition the set of *dynamically observable* contingent events $\mathcal{X}^{dyn}$ into two subsets (Figure 4.4):

- $\mathcal{X}^{vis}(\pi)$: events whose occurrence will always be *visible* to the agent, e.g., when the phone in my pocket rings;
- $\mathcal{X}^{hid}(\pi)$: events that are normally *hidden*, they become visible if specific actions are added to π, e.g., to observe that the water boils I need to be close and pay attention to it.

$$\textit{Contingent}\ (\mathcal{X}_{\mathfrak{ctg}}) \begin{cases} \textit{Invisible}\ (\mathcal{X}^{invis}) \\ \textit{Dynamically Observable}\ (\mathcal{X}^{dyn}) \begin{cases} \textit{Visible}\ (\mathcal{X}^{vis}) \\ \textit{Hidden}\ (\mathcal{X}^{hid}) \end{cases} \end{cases}$$

Figure 4.4: Classes of contingent points (ignoring *a priori known* ones).

Contingent events in a plan π are: $\mathcal{X}^{invis} \cup \mathcal{X}^{vis}(\pi) \cup \mathcal{X}^{hid}(\pi)$. Invisible events are intrinsic to a domain model, which has no means to observe them. However, the partition of observable events depends on the specification of a planning problem as well as on the

particular plan for that problem: an event initially in $\mathcal{X}^{hid}$ migrates to $\mathcal{X}^{vis}$ iff the actions needed to observe it are added to π. To make a plan controllable, an agent may need to perform additional sensing actions.

4.5.1 Needed Observations

Let $\mathcal{S}_\pi = (\mathcal{X}_{\text{ctl}},\ \mathcal{X}^{invis} \cup \mathcal{X}^{vis}(\pi) \cup \mathcal{X}^{hid}(\pi),\ \mathcal{R},\ \mathcal{C})$ be a POSTNU corresponding to a plan π without chained contingencies. $\mathcal{X}^{vis}(\pi)$ are the events initially specified as visible and those hidden that become visible because of the activity already planned in π. $\mathcal{S}_\pi$ can be in one of the following three cases:

(1) PO-Contr-Check$(X_C,\ \mathcal{X}^{vis}(\pi) \cup \mathcal{X}^{hid}(\pi),\ \mathcal{X}^{invis},\ \mathcal{R},\ \mathcal{C})$ is false: π cannot be made controllable even if all $\mathcal{X}^{hid}(\pi)$ is observed.

(2) PO-Contr-Check$(X_C,\ \mathcal{X}^{vis}(\pi),\ \mathcal{X}^{invis} \cup \mathcal{X}^{hid}(\pi),\ \mathcal{R},\ \mathcal{C})$ is true: π is controllable without any additional observation.

(3) Otherwise additional observation actions of events in $\mathcal{X}^{hid}(\pi)$ may make π controllable.

The third case requires identifying in $\mathcal{X}^{hid}$ events whose observation makes an augmented plan controllable. This is performed with procedure NeededObs (Algorithm 4), which searches a space of subsets of $\mathcal{X}^{hid}$. A path in the search tree corresponds to a set of events to move from the initial $\mathcal{X}^{hid}$ to $\mathcal{X}^{vis}$. A search state is expanded by nondeterministically choosing an event x to observe in a set of candidates. This is a backtrack point, but the order in which the candidates are examined is irrelevant. The search fails when no candidates can be found in $\mathcal{X}^{hid}$.

For finding one or all *minimal* sets of observation (in the set inclusion sense) that make the network controllable one can rely on breadth-first search. Because the order in which observation are considered is irrelevant, the procedure is extended to ignore any set of observations that was previously considered.

Algorithm 4 Finding a set of observations to make a Partially Observable STNU Dynamically Controllable

function NeededObs$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{invis}, \mathcal{X}^{vis}, \mathcal{X}^{hid}, \mathcal{R}, \mathcal{C})$
  if PO-Contr-Check$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{vis}, \mathcal{X}^{invis} \cup \mathcal{X}^{hid}, \mathcal{R}, \mathcal{C})$ **then**
    return $\varnothing$
  $\Sigma \leftarrow$ ObsCandidates$(\mathcal{X}^{hid})$ ▷ Selects a subset of $\mathcal{X}^{hid}$
  if $\Sigma = \varnothing$ **then**
    return *failure*
  $x \leftarrow$ NondeterministicallyChoose(Σ)
  $\sigma \leftarrow$ NeededObs$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{invis}, \mathcal{X}^{vis} \cup \{ x \}, \mathcal{X}^{hid} \setminus \{ x \}, \mathcal{R}, \mathcal{C})$
  if $\sigma =$ *failure* **then**
    return *failure*
  else
    return $\{x\} \cup \sigma$

ObsCandidates is a key function in NeededObs: it selects a subset of $\mathcal{X}^{hid}$ whose observation might render the POSTNU controllable. A naive version would simply try all possibly hidden events. Let us discuss how to find a focused set of candidates.

4.5.2 Relevant Candidates for Observation

The key insight to focus the search is to analyze what makes a given POSTNU not controllable. We first remark that the transformations in Table 4.2 simply make sure that the network stays consistent regardless of whether an invisible node occurs at its earliest or at its latest. More specifically, the first and second rules propagate the upper-case edges, while the last two rules propagate the lower-case edges of the invisible node.

We say that an edge e enforces the upper bound of a invisible point B if it was either: *(i)* introduced by the first or second rule of Table 4.2 with B as an invisible point; or *(ii)* derived through constraint propagation from an edge enforcing the upper bound of B. Similarly for the lower bounds with respect to the last two rules. We say that a bound of an invisible point is enforced by a sequence of edges if at least one edge of the sequence enforces that bound.

We make this information explicit in the network by labeling all edges with the set of bounds they enforce. For instance, if an edge is labeled with the set $\{A^{MAX}, B^{MAX}, B_{MIN}\}$, it means it enforces the upper bounds of A and B (noted A^{MAX} and B^{MAX}), and the lower bound of B (noted B_{MIN}).

Procedure PO-Contr-Check is extended to initialize these labels. Table 4.3 shows the extended transformations with the additional labels (in red). For an invisible event B, the added constraint inherits the label of the two constraints it originates from (the sets U and V) and has an additional projection B_{MIN} (resp. B^{MAX}) if it enforces the lower (resp. upper) bound of B.

Conditions	Added constraint
$A \xleftarrow{B{:}x,\ U} B \xleftarrow{y,\ V} C$	$A \xleftarrow{x+y,\ U \cup V \cup \{B^{MAX}\}} C$
$A \xleftarrow{B{:}x,\ U} B \xleftarrow{C{:}y,\ V} C$	$A \xleftarrow{C{:}x+y,\ U \cup V \cup \{B^{MAX}\}} C$
$A \xleftarrow{x,\ U} B \xleftarrow{b{:}y,\ V} C$	$A \xleftarrow{x+y,\ U \cup V \cup \{B_{MIN}\}} C$
$A \xleftarrow{a{:}x,\ U} B \xleftarrow{b{:}y,\ V} C$	$A \xleftarrow{c{:}x+y,\ U \cup V \cup \{B_{MIN}\}} C$

Table 4.3: Added constraints due to an invisible point B, with the labels propagated to track the bounds enforced by edges.

We also extend the classical STNU reduction rules of Table 4.1 as follow: if a reduction is triggered that produces an edge e_3 from two edges e_1 and e_2, then e_3 is annotated with the labels of both e_1 and e_2. Let φ be an STNU issued from the transformation by Table 4.3 of a POSTNU. If φ is not DC then it necessarily has a "culprit" sequence of edges (a semi-reducible negative cycle). This sequence, denoted φ_{Culprit}, enforces invisible and hidden nodes in $\mathcal{X}^{invis}$ and $\mathcal{X}^{hid}$; in the latter case it can give us observation candidates.

Proposition 4.5.1. *Let $\mathcal{S} = (\mathcal{X}_{\text{ctl}}\ \mathcal{X}^{invis} \cup \mathcal{X}^{vis} \cup \mathcal{X}^{hid},\ \mathcal{R},\ \mathcal{C})$ be a POSTNU without chained contingencies such that* PO-Contr-Check$(\mathcal{X}_{\text{ctl}}, \mathcal{X}^{vis}, \mathcal{X}^{invis} \cup \mathcal{X}^{hid}, \mathcal{R}, \mathcal{C})$ *is false, and φ be the corresponding STNU. If there is a sequence $\varphi_{Culprit}$ that enforces both the upper and lower bounds of events in $\mathcal{X}^{hid}$, then the observation of at least one such event is needed to make $\mathcal{S}$ controllable. Otherwise $\mathcal{S}$ cannot be made controllable.*

Proof. Making $\mathcal{S}$ controllable requires a change in φ in order to remove φ_{Culprit}. The only allowed changes result from observing events in $\mathcal{X}^{hid}$. Observing an event A means that edges labeled with A_{MIN} or A^{MAX} (and only those) won't be introduced in φ. Since the only edges that can be removed are the ones labeled with a node in $\mathcal{X}^{hid}$, then if there is

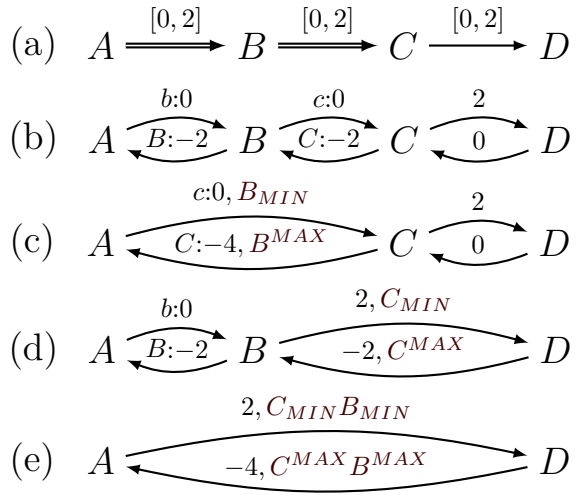


Figure 4.5: (a) A POSTNU with two contingent points B and C; its labelled distance graph (where some requirement edges are omitted for clarity) in different observability cases: (b) both B and C are observable, (c) B is invisible, (d) C is invisible, (e) both B or C are invisible.

none (φ_{Culprit} enforces only nodes in $\mathcal{X}^{invis}$) φ remains non DC regardless of additional observations. Otherwise the observation of at least one event in $\mathcal{X}^{hid}$ enforced by the sequence φ_{Culprit} is needed.

Let φ' be a partial projection of φ defined as follows: for every contingent link $A \overset{[l,u]}{\Longrightarrow} B$ where B has its lower (resp. upper) and only its lower (resp. upper) bound enforced by φ_{Culprit}, we replace in φ' this link by a new link $A \overset{[l,l]}{\Longrightarrow} B$ (resp. $A \overset{[u,u]}{\Longrightarrow} B$). φ' is essentially a version of φ where the duration of all contingent links with only one bound enforced by φ_{Culprit} is known in advance.

The main feature of φ' is that it contains all edges of φ_{Culprit}, making φ' not DC. In φ', complete knowledge of events with just one enforced bound is not sufficient to remove the inconsistent sequence. Consequently, removing the inconsistency requires at least one event with its two bounds participating in φ_{Culprit} to be observed. □

The ObsCandidates function returns $\varnothing$ when there is a sequence φ_{Culprit} (obtained when PO-Contr-Check is false) which does not enforce the upper and lower bounds of a node in $\mathcal{X}^{hid}$. Otherwise it returns all such nodes.

Figure 4.5(e) gives an example of such an inconsistent network. It was built by applying PO-Contr-Check to the network of Figure 4.5(a) with B and C marked invisible. The edge $A \xrightarrow{2,\ C_{MIN}B_{MIN}} D$ states that the lower bounds of both B and C require a delay from A to D of at most 2. The edge $A \xleftarrow{-4,\ C^{MAX}B^{MAX}} D$ states that the upper bounds of both B and C require a delay from A to D of at least 4. These two edges will be detected as a culprit cycle that enforces both the lower and upper bounds of B and C. To make the network controllable, at least one of B or C must be observed. This would result in the STNUs of either Figures 4.5(c) and 4.5(d) which are both controllable.[2]

[2]Note that the network in Figure 4.5a has no chained contingencies.

4.6 Planning with a POSTNU

Our proposed approach and procedures PO-Contr-Check and NeededObs can be integrated and used incrementally into our temporal planner. For this purpose, partial plans are extended to contain contingents events together with conditions for their observation. The Temporal Constraint Network is adapted to reason with such events and the planning algorithm considers a new kind of flaw that detects when a plan is uncontrollable and imposes some observations to enforce controllability of the plan.

4.6.1 Representation of Contingent Events

To handle POSTNUs, we need a way to refer to contingent events and their observation conditions. For this purpose, our definition of a chronicle (Section 2.4) is extended to allow the presence of contingent timepoints and contingent constraints. Furthermore, every observable contingent event e is associated with observation conditions expressed as a pair $(\mathcal{F}_e, \mathcal{C}_e)$ where $\mathcal{F}_e$ and $\mathcal{C}_e$ are assertions and constraints denoting a set of sufficient conditions for the observation of e. For instance, consider the event e_{cooked} that the dinner is cooked. We need to observe that event in order to avoid overcooking. Such an event would be associated with the conditions:

$$(\{\, [e_{cooked}]\ \texttt{location} = \texttt{Kitchen}\,\}, \varnothing)$$

that requires me to be in the kitchen at the occurrence time of e_{cooked}.

A partial plan (i.e. a chronicle ϕ), is associated with a set of contingent events $\mathcal{X}_{\text{ctg}} = \mathcal{X}^{invis} \cup \mathcal{X}^{vis}(\phi) \cup \mathcal{X}^{hid}(\phi)$ where:

- $\mathcal{X}^{invis}$ is the fixed set of events that can never be observed. We assume them to be given in the domain definition.
- $\mathcal{X}^{vis}(\phi)$ contains all points whose occurrence will be observed. The initial set of visible contingent (i.e. $\mathcal{X}^{vis}(\phi_0)$) is composed of all dynamically observable events who are unconditionally observed (i.e. any event e such that $\mathcal{F}_e = \mathcal{C}_e = \varnothing$).
- $\mathcal{X}^{hid}(\phi_0)$ contains all dynamically observable contingent events that we have not proved to be observable yet. Initially, any event that is not unconditionally observed is assumed to be hidden (i.e. in $\mathcal{X}^{hid}(\phi_0)$).

4.6.2 Main Temporal Constraint Network

FAPE planning algorithm heavily relies on the availability of temporal knowledge regarding the current partial plan. To maintain completeness of the planner, we need every temporal constraint in a plan ϕ to be sound (i.e. to be present in any solution obtained by refining ϕ).

For this purpose, the consistency of the temporal network is checked by assuming all hidden points to be visible using:

$$\text{PO-Contr-Check}(\mathcal{X}_{\text{ctl}},\ \mathcal{X}^{vis}(\phi) \cup \mathcal{X}^{hid}(\phi),\ \mathcal{X}^{invis},\ \mathcal{R},\ \mathcal{C})$$

The result is optimistic: some currently hidden points could indeed remain hidden. The DC-Check algorithm used internally is the one of Morris [Mor14] that runs in $O(n^3)$ where n is the number of timepoints in the network.

This consistency check allows to prune dead-ends due to detected inconsistencies. However, the planner still need to query the temporal network and the cubic algorithm for DC-Check does not compute the all-pairs shortest paths required to efficiently query the network. DC checking procedures that rely on such a minimal network are typically too expensive for considering an online use for planning (e.g. the algorithm of Morris, Muscettola, and Vidal [MMV01] is pseudo-polynomial). Instead, we use an STN $\mathcal{S}$ composed of all requirements constraints of the POSTNU. Any time a new requirement is inferred by PO-Contr-Check, the corresponding constraint is added to $\mathcal{S}$. $\mathcal{S}$ is maintained minimal by an incremental Floyd-Warshall procedure (see Section 3.2.3.1) and can be used to answer the queries of the planner in constant time.

4.6.3 Planning for Observation

We now present how the planner is extended to reason on observability. The core of this mechanism is a new *flaw detection function* that detects when new observations are needed and proposes sets of observations that would render the current plan controllable.

Uncontrollable Plan Flaw. We define a new type of flaw denoting an *uncontrollable plan*. This flaw is detected if the current POSTNU is not controllable given that all currently hidden events remain hidden. The plan is thus *uncontrollable* if the procedure

$$\text{PO-Contr-Check}(\mathcal{X}_{\text{ctl}},\ \mathcal{X}^{vis}(\phi),\ \mathcal{X}^{invis} \cup \mathcal{X}^{hid}(\phi),\ \mathcal{R},\ \mathcal{C})$$

returns false.

Resolvers for an Uncontrollable Plan. Given an uncontrollable plan ϕ, one must find a set of observations that would make it controllable. This is done using NeededObs to find all minimal sets of observations $\{ O_1, \ldots, O_N \}$ where each O_i is such that:

- O_i is a set of hidden events, i.e., $O_i \subseteq \mathcal{X}^{hid}(\phi)$
- the observation of all events $e \in O_i$ would make the plan controllable, i.e., PO-Contr-Check$(\mathcal{X}_{\text{ctl}},\ \mathcal{X}^{vis}(\phi) \cup O_i,\ \mathcal{X}^{invis} \cup \mathcal{X}^{hid}(\phi) \setminus O_i,\ \mathcal{R},\ \mathcal{C})$ returns true.

The *uncontrollable plan* has N resolvers, one for each set of observations.

For a given set of observation O_i, the resolver is a plan restriction $\phi \xrightarrow{(\mathcal{F}_{O_i}, \mathcal{C}_{O_i})}_R \phi'$ where $\mathcal{F}_{O_i} = \bigcup_{e \in O_i} \mathcal{F}_e$ and $\mathcal{C}_{O_i} = \bigcup_{e \in O_i} \mathcal{C}_e$. Essentially, this ensure that the condition for the observation of any $e \in O_i$ are met in all plans refined from ϕ'. This is made explicit by transferring all events in O_i from the hidden set to the visible set of ϕ' (i.e. $\mathcal{X}^{vis}(\phi') \leftarrow \mathcal{X}^{vis}(\phi) \cup O_i$ and $\mathcal{X}^{hid}(\phi') \leftarrow \mathcal{X}^{hid}(\phi) \setminus O_i$).

This flaw detection mechanism allows to detect, while planning, whether the current partial plan is controllable. If not, then the planner will consider possible sets of observations, each of which would render the partial plan controllable. This is not done by directly considering sensing actions but instead by stating condition on the state of the world. Supporting those conditions will typically trigger the insertion of new actions into the plan.

4.6.4 Dispatchable and Structural Timepoints

So far we have avoided another restricting assumption in the classical definition of an STNU which is that all controllable timepoints must be dispatched: given a controllable timepoint $x \in \mathcal{X}_{\text{ctl}}$, a dynamic execution strategy must assign it an execution time t no later than t.

Let us consider a simple example to see how this can be problematic in the case of rich temporal models. We use a version of our move action from Chapter 2 extended to have an uncertain duration:

$\mathsf{move}(r, d, d')$	
task:	$\mathtt{move}(r, d, d')$
dependent:	no
assertions:	$[t_{start}, t_{end}]\ \mathtt{loc}(r) : d \mapsto d'$
	$[t_{start}, t]\ \mathtt{free}(d) : \bot \mapsto \top$
	$[t', t_{end}]\ \mathtt{free}(d') : \top \mapsto \bot$
subtasks:	$\varnothing$
constraints:	$\mathit{connected}(d, d') = \top$
	$t_{start} \xrightarrow{[15,20]} t_{end}$
	$t = t_{start} + 1$
	$t' = t_{end} - 1$

Figure 4.6: A move action with uncertain duration. Moving from location d to location d' takes between 15 and 20 time units.

Using this move action, suppose we want to bring a rover r from a location d1 to a location d2, where d2 only become *free* after time 30. Given a temporal origin $\mathcal{O}$, a solution to this problem would be to have an action $\mathsf{move}(\mathtt{r}, \mathtt{d1}, \mathtt{d2})$ to be scheduled after $\mathcal{O}$ while ensuring that its t' timepoint occurs at least 30 time units after $\mathcal{O}$. The POSTNU corresponding to this solution is given in Figure 4.7.

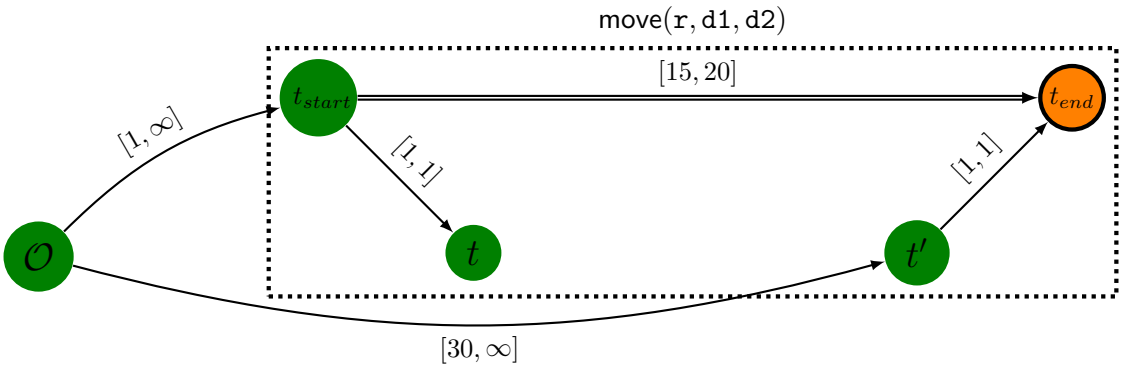


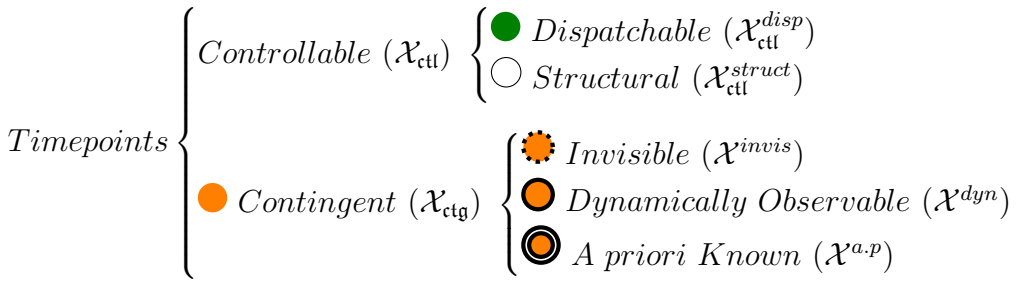
Figure 4.7: STNU of our moving problem.

However this STNU is not controllable: t' can not be dispatched exactly one time unit before the occurrence of t_{end} as it would require knowing in advance when our move action finishes. Our planning problem however has an intuitive solution: if we schedule the action to start at time 16, then it will not end before time 31 and we are guaranteed that d2 be free when needed.

The difference lies in how t' is considered. In our intuitive solution, t' is simply meant to express constraints relating $\mathcal{O}$ and t_{end}, here to condition that at least 31 time units elapse between $\mathcal{O}$ and t_{end}. In fact, we never require a value to be given to t', its domain will instead be reduced due to the occurrence of other events such as t_{start} and t_{end}. We say that t' is a *structural timepoint*: it is simply meant to express requirements about other timepoints. While the reasoning on t' is simple enough, temporal action can in general contain many such structural timepoints involved in more complex chains of constraints.

Definition 4.6.1 (POSTNU with Structural Timepoints)**.** We define a POSTNU with Structural Timepoints as a POSTNU $\mathcal{S} = (\mathcal{X}_{\text{ctl}}, \mathcal{X}_{\text{ctg}}, \mathcal{R}, \mathcal{C})$, where the set of controllable points $\mathcal{X}_{\text{ctl}}$ is partitioned into:

- a set of *dispatchable timepoints* $\mathcal{X}_{\text{ctl}}^{disp}$, and
- a set of *structural timepoints* $\mathcal{X}_{\text{ctl}}^{struct}$ that are only involved in requirement constraints.



We do not give a complete definition of the controllability of such a network but instead remark that this definition can be built by simply restricting a schedule to provide an execution time for dispatchable points only (while our previous definition considered it for all controllable timepoints). The result is that *(i)* no decision must be made regarding the time at which *structural points* are dispatched *(ii)* the STN network resulting from the execution must still be consistent. The latter condition enforces that all structural timepoints have a valid instantiation.

Definition 4.6.2 (Controllability of POSTNU with Structural Points)**.** We say that a POSTNU with Structural Timepoints is controllable if it has a viable dynamic execution strategy whose schedule is restricted to dispatchable timepoints.

Given this definition of POSTNU with Structural Timepoints, we would be interested in knowing whether it is controllable and find an execution strategy for it. We now demonstrate how a POSTNU with Structural Timepoints can be compiled into an equivalent POSTNU in which all structural timepoints have been removed.

Algorithm 5 performs this compilation by first considering an STN $(\mathcal{X}, \mathcal{R}^{struct})$ that contains all timepoints $\mathcal{X}$ of a POSTNU with structural points and a restricted set of requirement links $\mathcal{R}^{struct} \subseteq \mathcal{R}$ that contains all links whose start or end point is a structural timepoint. An All-Pairs Shortest Path (APSP) procedure is used to compute the minimal network of this STN: $(\mathcal{X}, \mathcal{R}^{min\text{-}struct})$. While, the original set of requirements $\mathcal{R}^{struct}$ only contained constraints involving at least one structural point, $\mathcal{R}^{min\text{-}struct}$ now contains minimal constraints that can relate two contingents or dispatchable points. The final POSTNU is built by taking only those constraints that do not involve any structural

points both from the original constraint set ($\mathcal{R}$ and $\mathcal{C}$) and from the minimal network ($\mathcal{R}^{min\text{-}struct}$).

The resulting POSTNU does not contain any structural timepoints and one can use the customary algorithms for checking its controllability (e.g. PO-Contr-Check).

Algorithm 5 Compilation of structural timepoints

Input:
A POSTNU with structural timepoints
Output:
An equivalent POSTNU with no structural timepoints

function CompileStructuralPoints($\mathcal{X}_{\texttt{ctl}}^{disp} \cup \mathcal{X}_{\texttt{ctl}}^{struct}, \mathcal{X}_{\texttt{ctg}}, \mathcal{R}, \mathcal{C}$)
$\mathcal{X} \leftarrow \mathcal{X}_{\texttt{ctl}}^{disp} \cup \mathcal{X}_{\texttt{ctl}}^{struct} \cup \mathcal{X}_{\texttt{ctg}}$
$\mathcal{R}^{struct} \leftarrow \{\, c \,|\, c \in \mathcal{R} \wedge c$ involves a structural timepoints $x \in \mathcal{X}_{\texttt{ctl}}^{struct} \,\}$
$(\mathcal{X}, \mathcal{R}^{min-struct}) \leftarrow \text{APSP}(\mathcal{X}, \mathcal{R}^{struct})$
if $\mathcal{R}^{min\text{-}struct}$ has a negative cycle **then**
return *failure*
$\mathcal{R}^{no-struct} \leftarrow \{\, c \,|\, c \in (\mathcal{R} \cup \mathcal{R}^{min-struct}) \wedge c$ involves no structural timepoints $\,\}$
return $(\mathcal{X}_{\texttt{ctl}}^{disp}, \mathcal{X}_{\texttt{ctg}}, R^{no-struct}, \mathcal{C})$

Theorem 4.6.1. *Algorithm 5 is sound and complete.*

Proof. We first make the observation that Algorithm 5 only adds constraints that are inferred through an APSP algorithm. Hence all constraints added to the compiled POSTNU are correct.

Now we prove that if the STNU is controllable, then all structural timepoints have a consistent instantiation. Assume that, after execution, the STN with structural timepoints would contain an inconsistency that is not present in the compiled one. This means that there is a cycle of negative length involving at least one structural timepoint T: $A \xrightarrow{[-,x]} T \xrightarrow{[-,y]} B \xrightarrow{[-,z]} A$ such that $x + y + z < 0$. To show that the compiled STN would have the same inconsistency, we rely on the fact that APSP preserve minimal distances. Indeed, in this case, APSP would have inferred an $A \xrightarrow{[-,x+y]} B$ constraint and the negative cycle would also appear in the compiled STN. In the case where all timepoints of a negative cycle are structural, the inconsistency would have been detected in the compilation phase by a negative cycle in $\mathcal{R}^{min\text{-}struct}$. □

In the special case where a structural timepoint t_1 is rigidly fixed to another timepoint t_2 through a constraint $t_1 \xrightarrow{[d,d]} t_2$, a simpler procedure can be applied to remove it from the network (see the compilation of rigid components of an STN by Tsamardinos, Muscettola, and Morris [TMM98]). This is for instance the case of the timepoint t' in Figure 4.7. This technique is however not applicable in general as an action might contain many structural timepoints that are not rigidly constrained such as the timepoints associated to an unrefined task.

Example 4.3. We illustrate our compilation of structural timepoints on a slightly more complex example introduced by Rabideau et al. [Rab+99] in the context of STN that involves warming up an instrument some time before using it. We define an action

take_measure that requires *(i)* that an instrument i had been warming up for at least 10 seconds *(ii)* that the instrument be idle for the duration of the action.

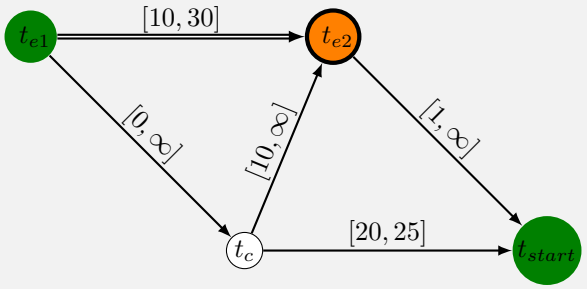
$$
\begin{array}{rl}
\textsf{take\_measure}(i) & \\
\text{task:} & [t_{start}, t_{end}]\texttt{take\_measure}(i) \\
\text{assertions:} & [t_c, t_c + 10]\ \texttt{state}(i) = warming \\
& [t_{start}, t_{end}]\ \texttt{state}(i) = idle \\
\text{constraints:} & t_{start} - 25 \leq t_c \leq t_{start} - 20
\end{array}
$$

The chronicle below defines two change assertions. $e1$ is a change of the state of an instrument i from *idle* to *warming* at time t_{e1}. $e2$ is a change of the state of an instrument c from *warming* to *idle* at time t_{e2}. The timepoint t_{e2} is contingent and occurs between 10 and 30 seconds after t_{e1}

$$
\begin{array}{l}
(\{\varnothing, \\
\ \{[t_{e1} - 1, t_{e1}]\ \texttt{state}(i) : idle \mapsto warming, \\
\ \ [t_{e2}, t_{e2} + 1]\ \texttt{state}(i) : warming \mapsto idle\}, \\
\ \{t_{e1} \xrightarrow{[10,30]} t_{e2}\})
\end{array}
$$

By making $e1$ and $e2$ respectively support the first and second statement of the take_measure action, we would obtain the following STNU where:

- t_c must occur once the instrument is warming (i.e. after t_{e1})
- t_c must occur at least 10 seconds before it switches back to idle (i.e. at least 10 time units before t_{e2}).
- t_{start} must occur when the instrument is idle (i.e. strictly after t_{e2})
- t_{start} must occur between 20 and 25 seconds after t_c.



If we assumed t_c to be a *dispatchable* timepoint, the STNU would not be controllable: t_c would be constrained to happened directly after t_{e1} (to make sure the 10 seconds delay with t_{e2} is respected). If t_{e2} occurs 30 seconds after t_{e1} there would be no possible instantiation of t_{start}.

Taking a step back, we can see that this problem should have a solution. By ensuring that t_{start} occurs at least 20s after t_{e1} and at most 15s after t_{e2} we are guaranteed to find a solution where the warming condition holds. The reason why the proposed STNU is not DC comes from a too restrictive assumption: the t_c timepoint

does not need to be dynamically instantiated. It is sufficient to show that it will have at least one valid value regardless of the choice made for other timepoints.

This can be modeled by considering t_c as a structural timepoint. The figures below give the intermediate step and the result of compiling the structural timepoint t_c of this network using Algorithm 5.

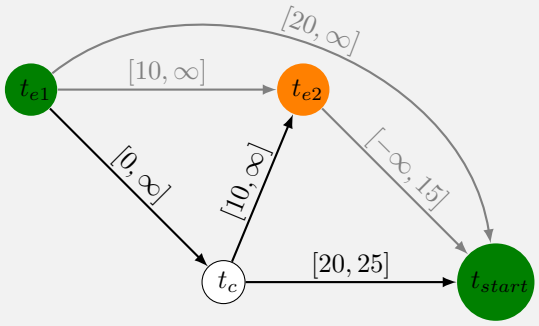


Figure 4.8: In black are the constraints of the original POSTNU that are used as a base for the APSP computation (i.e. $\mathcal{R}^{struct}$). In gray are the constraints inferred with the APSP step (i.e. $\mathcal{R}^{min\text{-}struct} \setminus \mathcal{R}^{struct}$).

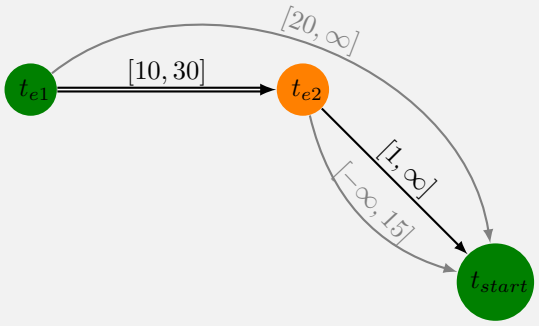


Figure 4.9: The compiled POSTNU limited to *contingent* and *dispatchable* timepoints. In black are the constraints originally present. The constraints in gray are those introduced after the APSP step on t_c.

4.7 Empirical Evaluation

For evaluating our approach, we extended the ANML language for representing contingent events and their observation conditions. For illustration, we encoded our running example of cooking dinner with an additional action allowing the agent to request his partner to notify him when leaving a location. The corresponding problem can be found in Section B.4. FAPE was able to find a solution to this problem by:

- scheduling a cooking action,
- recognizing that the partial plan is not controllable and that the observation of the `wife_driving` event would make the plan controllable,
- adding the conditions for observation of `wife_driving` to the partial plan,
- scheduling a notification request while `wife` is at `work` to make those conditions hold in the solution plan.

Finding this plan presented no difficulty for FAPE that took well under a second to solve the problem.

In order to assess the various components involved in computing a plan with partial observability and sensing actions, we constructed an artificial logistics domain as follows. A domain has n areas. In each area several contingent events of different types occur. An event e produces an effect that holds for a given amount of time. An agent can: *(i)* move between the areas, *(ii)* observe the occurrence of events in its area, and *(iii)* handle an

event by acting during the period its effect holds. Each problem instance has two agents and few tasks corresponding to events that need be handled.

The planner is run on 192 randomly generated instances of this domain, each problem containing from 2 to 20 areas and from 8 to 200 events. Each task typically requires a few move actions and one handling action for the targeted event. Furthermore, handling an event leads either to: *(i)* add a sensing action to observe this event; *(ii)* add one or several sensing actions to observe earlier events to provide enough information; or *(iii)* no sensing action is needed. The planner is able to identify which of these options is possible and choose a desirable one appropriately.

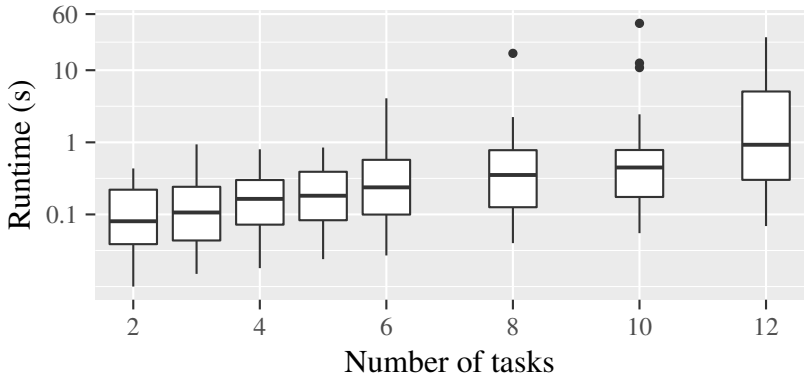


Figure 4.10: FAPE runtime distribution on 192 random instances of a logistics domain with 8 to 200 contingent events (isolated points are beyond the maximum plotted by R).

Figure 4.10 shows the runtime distribution to plan problems having different numbers of tasks. More tasks result in more actions in the final plan, which naturally increases planning time. The planner spent on average 47.31%, and at most 68%, of its processing time on POSTNU related code (that is, checking controllability and looking for needed observations). Three problems with 12 tasks were not solved for a timeout of 60 seconds. On those problems, the limitation was the heuristic of the planner that failed to provide sufficient guidance. Indeed, our usual heuristics are not designed to assess the specifics of partial observability and the resulting additional sensing actions. Beyond the quantitative measures, it is important to note that, with a reasonable overhead, the planner was able to incrementally identify the necessary observations to keep its plan dynamically controllable. More importantly, this was done while maintaining and extending the causal structure of the plan to support the required sensing actions.

We further use the POSTNUs encountered while planning on those examples to explore the performance of NeededObs. Figure 4.11 shows an empirical evaluation of the performance of NeededObs on those POSTNUs corresponding to partial plans in our test domain with a few hundred nodes and up to 200 contingent events. The labeling technique speeds up the search by several orders of magnitude over the naive approach. Indeed, NeededObs required on average 5 iterations, and as many calls to DC-Check, to find a minimal set of observations. Even on large networks, this number never exceeded 13.

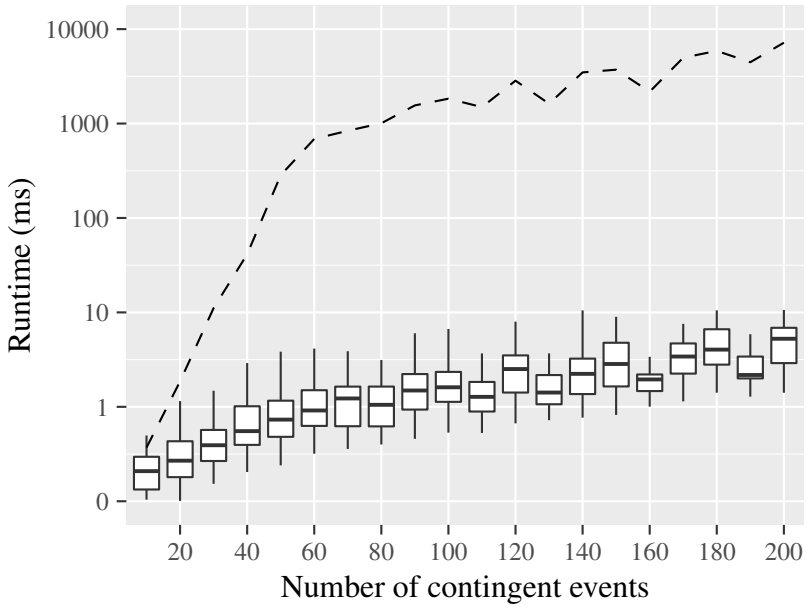


Figure 4.11: NeededObs runtime distribution on 2264 networks of 32 to 311 events in total, requiring 1 to 4 observations to be controllable (median, first and third quartiles, max and min). The dashed line gives the median runtime when our labeling technique is not used to extract a restricted set of candidate observations.

4.8 Conclusion

In this chapter, we presented a fairly comprehensive study of temporal networks with partially observable contingent events. The dynamic controllability of STNUs assumes all contingent events to be observable. In many domains they are not. However, the agent can observe them if he takes adequate sensing actions, hence if he plans for those actions.

We proposed a transformation of POSTNUs into STNUs to determine their controllability using known DC-Check algorithms. We formally proved PO-Contr-Check to be sound; it is complete for an interesting subclass of networks.

Building on PO-Contr-Check, we proposed NeededObs for finding a minimal set of observations that makes a POSTNU controllable. We developed an edge labelling mechanism that focuses NeededObs and demonstrated formally its main property. Empirical tests show that NeededObs remains efficient even on networks corresponding to large plans.

Finally, we showed how the proposed approach can be integrated incrementally in FAPE. We proposed a new flaw detection function that identifies the need for additional observations and imposes the necessary conditions for maintaining a plan controllable. The planner further relies on dedicated constraint networks for handling contingent events and structural timepoints.

The essential contribution of this chapter is to consider visibility as a dynamic property over which an agent has some control. On the contrary, classical models of partially observable dynamic systems consider only two static categories of variables: invisible and observable.

Chapter 5

Online Planning and Acting with FAPE

Contents

5.1 Introduction

So far, we have been interested in solving offline planning problems. In this chapter, we are interested in adapting the algorithms previously presented to handle robotics planning tasks that require online deliberation in order to act in the real world. Autonomous agents and robotics in particular present many use cases where the planning procedure we have presented would not be useful if limited to an offline setting.

Indeed, our planner assumes a fairly abstract view of the problem of actuating a robotic platform. For acting in the real world, those abstract actions must eventually be refined to commands understandable by the robotic platform. Such refinement is often non trivial and requires further deliberation, e.g., for choosing how to grasp an object. Furthermore, in open environments, one cannot assume that a plan will be carried out entirely without encountering failures. In such cases, the execution of a plan must be monitored and adapted online to handle unforeseen contingencies.

In this chapter, we discuss the integration of our planning system in a more general architecture aiming at deliberate acting in robotics. For this purpose, we will study a service robotics domain where a robot must accomplish several tasks such as tidying up an apartment and bringing items to a human user, possibly with some temporal constraints.

An agent has a model of the environment and of its own capabilities expressed in particular as a set of chronicles. The agent further has perception capabilities that provide a partial view of the evolution of the world and actuation capabilities provided as primitive commands of a robotic platform. The agent should, through deliberate planning and acting, achieve a set of objectives expressed as tasks or assertions on the state of the world. For this purpose, the agent should find and execute a plan that achieves its objectives while taking into account the modeled and observed contingencies and the partial knowledge of the environment. Executing the plan involves refining the actions into primitive commands and monitoring the overall process to make sure the evolution of the world matches the agent's expectation. In case of discrepancies that would prevent the agent from achieving its objectives, its current plan should be adapted to go back to a nominal behavior.

5.2 Related Work

The needs of deliberate action in robotics goes well beyond the usual planning problem as the execution of a symbolic plan typically involves *(i)* refining symbolic actions down to primitive commands, *(ii)* monitoring the execution of the plan in a partially observable environment, and *(iii)* reacting to unforeseen contingencies by adapting its behavior. We here give an overview of some of the deliberation capabilities that a robotic agent must have in order to act in the real world. A comprehensive view of those functions is given by Ingrand and Ghallab [IG14].

Monitoring. Execution monitoring is the process of identifying the discrepancies between the observations of the real world and predictions derived from an agent's representation of the world. In addition to a *fault detection* function, execution monitoring is typically used for *fault isolation* and *fault diagnosis*, that is, identifying where the problem is situated and providing an explanation for it.

Execution monitoring has been well studied in the context of industrial control, where systems have very limited autonomy. A survey of related monitoring systems, targeting

the sensory-motor level of autonomous robots, is given by Pettersson [Pet05].

Model-based diagnosis was used for monitoring and recovery in the Remote Agent Experiment (DS1/RAX) [Mus+98; Ber+00]. Each component of the spacecraft is associated with a graph were nodes denote the nominal and failure states and edges describe nominal transitions (associated to a command) or exogenous transitions (denoting failures). The latter are associated with transition probabilities that are used to find out the most likely transitions that match the observations. The system is also capable of recovery by searching for a set of nominal transitions such that all components return a nominal state. This system is integrated with the rest of the deliberation modules and was demonstrated to be effective for a spacecraft.

Several approach extends planning operators to facilitate the monitoring of a plan. Doherty, Kvarnström, and Heintz [DKH09] provide an integration of planning and monitoring using Temporal Action Logic (TAL). The system uses the model of actions to generate control formulas using an annotated subset of an action conditions and effects. The control formula is used to detect discrepancies at acting time regarding both the individual actions and the causal structure of the plan.

Fichtner, Großmann, and Thielscher [FGT03] extend the usual planning operators with abnormal conditions representing conditions which, when not true, can cause an action failure. Those conditions are by default assumed to hold. When discrepancies are detected, the system uses a prioritized default logic to find out the most likely cause for failure, i.e. one or several abnormal conditions that do not hold.

Fraser, Steinbauer, and Wotawa [FSW05] extend a planning problem with plan invariants in the form of a logical sentence that should hold at any point of a solution plan. Monitoring is done by checking that neither plan invariants nor causal links are violated during execution.

Recovering. Regardless of the system used for detecting and explaining discrepancies, a complete acting system must be able to recover from a discrepancy by adapting its behavior. In the context of plan-based acting systems, this is generally achieved by changing the plan by either repairing the current plan or replanning entirely.

A metric of plan stability is defined by Fox et al. [Fox+06] as a function of the number of actions that differ from the original plan in the new one. The authors argue that stability is an important property as it *(i)* reduces the cognitive load on a human operator supervising the robotic system, and *(ii)* allows for a better prediction of the action sequences that will be executed and facilitates the transitions between those actions. Similarly, Cushing and Kambhampati [CK05] argue that the metric for evaluating the quality of a plan should take into account the commitments previously made by the acting system.

A practical setting for maintaining plan stability as well as limiting the computational load of the system is to favor plan repair in place of replanning [Fox+06]. A popular framework for plan repair is to use local search at the core the planner's algorithm. Indeed, local search allows both the removal of problematic actions or constraints as well as extending the plan to fulfill unsatisfied conditions. This approach is notably used in GPG [GS00], LPG [GSS03] and ASPEN [Chi+00a] for both planning and repair. Bajada, Fox, and Long [BFL14] propose a local search mechanism for repairing or optimizing a temporal plan subject to various quality metrics.

Various system handle plan repair by removing parts of the plan to come up with a partial plan from which search will be started. Krogt and Weerdt [KW05] use this tech-

nique in an extension to VHPOP [YS03]. Their planner uses classical planning heuristics to decide which constraints or action to remove in a first *de-refinement* process. The resulting plan is used as a starting point for search. If the search fails to provide a solution, their planner will try to remove more actions until eventually all actions are removed which is equivalent to replanning. A similar approach is used by Boella and Damiano [BD02] in the context of HTN planning: the system will first try to remove the first inapplicable action. If the system fails to find a plan, then the planner will remove its parent in the decomposition tree, together with all its subactions. The process is continued until finding a plan or reaching the root of the task network. POCL planners, including most constraint based planners, naturally support this repair scheme that was for instance demonstrated with IxTeT [LI04; Lem04] or HiPOP [Bec16].

Another possibility is to replay the reasoning that led to the original plan. This generally involves storing the reason for which an action was inserted and remembering which choice always leads to inconsistent plans. The planner can then replay the reasoning from an updated state, avoiding some dead-ends and only inserting actions whose insertion is still required. This is the approach taken by RepairSHOP [War+07] and PANDA [BSB08] in the context of hierarchical planning.

Perceiving. Perception is another critical capability for robotic systems. While the problem of localization of a robotic agent as been successfully handled by Simultaneous Localization and Mapping techniques (SLAM) [Mon+03; BD06], other perception problems remain more challenging.

One of those is the *anchoring* problem of relating symbols and perception data that refer to the same physical object [CS02]. Anchoring is a critical capability as planning, and other deliberation functions, reason on symbolic attributes that must eventually be related to the actual properties of physical objects they represent (e.g. for the purpose of monitoring a plan execution). While much work has been dedicated to the classification of an object from visual data (e.g. recognizing that a given object is a cup), only a few systems have been focused on symbol anchoring (e.g. [CS02; Kar+08; HKD10]).

An additional problem is the one of situation and event recognition from a set of temporally situated observations. Chronicle recognition techniques encode a possible situation in the form a chronicle. The set of possible situations is filtered online by eliminating situations that do not match the observation made so far [DGG93; DL07; Gha96; HKD10]. In addition, several techniques for activity recognition draw from the related work in signal processing and plan recognition, as extensively surveyed by Krüger et al. [Krü+07].

Dispatching. Plan dispatching is the problem of dynamically deciding when to start each action of the plan. The problem is trivial for non-temporal planning as one must simply start an action once the preceding one has finished. In temporal planning, each action is associated with a start timepoint t_{start} whose associated constraints form a temporal network. An action can be started a time t if $t_{start} = t$ is a consistent assignment to its start timepoint. To carry out the execution of a plan, a dispatching algorithm must keep track of the assignments made to all past timepoints and propagate their consequence in the rest of the temporal network. Various efficient algorithms have been proposed to perform this propagation for Simple Temporal Networks (e.g. [TMM98; MMT98]).

The problem is more complex with the Simple Temporal Network with Uncertainty (STNU) as one must account for contingent timepoints in addition to controllable ones.

An STNU is dynamically controllable if controllable timepoints can be dynamically dispatched by looking at past observations only. Because of this property, most algorithms for checking the dynamic controllability of an STNU make explicit the constraints necessary for deciding whether a controllable timepoint can be executed at the present time [NKD15; NKD14a; NKD13; Mor14]. Additional algorithms have been proposed to provide a more efficient online propagation of dynamically controllable STNU [Hun16; Hun10]. Beyond these constraint based approaches, another possibility is to translate an STNU to a Timed Game Automata (TGA) and rely on established model checking tools to provide a execution strategy [Vid00; Cim+14b; Cim+16; MO15].

Those approaches assume a central controller that is instantly aware of all decisions regarding the execution of controllable timepoints. This assumption does not hold in the context of multi-agent plans where communication between agents is not always possible. For this purpose, Casanova, Pralet, and Lesire [CPL15] propose an algorithm for managing the execution of a Multi-agent STN (MaSTN [BD13]) where each agent has a partial view of another agent's activities. The author consider several communication scenarios ranging from complete observability by a central supervisor to local information exchange between agents. Mogali, Smith, and Rubinstein [MSR16] propose an algorithm that decouple the activities of all agents involved by a MaSTN by an offline distributed preprocessing. In the same line, Casanova et al. [Cas+16] consider a Multi-agent STNU (MaSTNU) problem that involves uncertain durations. The authors propose an offline technique that compiles a MaSTNU into one SNTU for each agent that does not require any knowledge on the activity of other agents.

Acting. Acting is the problem of refining an action down to primitive commands executable by the platform. A tempting approach would be to simply map the planned actions to commands of the system as it was demonstrated on Shakey with the PLANEX system [Fik71]. This scheme however quickly finds its limits as the planned actions typically remain to abstract to handle the physics of the real world.

Procedure-based approaches rely on hand written rules to decide how to refine an action based on the current context. RAP [Fir87] and PRS [Ing+96] are two systems that rely on *operational procedures* (or skills) to achieve goals or react to events. Both systems commit to their goals, possibly falling back to a different skill if the one initially chosen fails to achieve the goal at hand. XFRM [BM94] and TCA/TDL [Sim92; SA98] are other influential examples of procedure based acting systems. The former uses rules to transform a current plan in plan space while the latter use a task decomposition framemork for this purpose. While some of those system are endowed with limited planning capabilities (e.g. by simulating the execution of a given skill), they have mainly be used in conjunction with task planners. PRS has been integrated with SIPE [Wil90] in Cypress [WM95] and CPEF [Mye99], RAP has been integrated with PRODIGY [VR98] and AP [Bon+97] and TDL has been integrated with Casper [Chi+00a].

RMPL (Reactive Model-based Programming Language [IRW99; LW14; EWH10; KWA01] provides a comprehensive representation for planning and acting by combining a model of the system as a state transition system (encoding nominal and failure transitions) as well as reactive programming constructs encoding the control rules. An RMPL model is translated into Temporal Plan Networks that represent a set of possible plans and their associated decision nodes in an extension of STNs. Planning with a TPN can be reduced to associating a choice to each decision node so that the system takes a desirable path. Extension to Temporal Plan Networks have been devised to

handle probabilistic domains [SW14] or uncertain domain [LW14].

IDEA [Mus+00; Mus+02] and T-ReX [PRM10; McG+08; McG+09] both provide a way of combining several components with heterogeneous tasks (e.g. long term planning or navigation control) through a set of shared timelines. The EUROPA planner [Bar+12] (or possibly APSI [Ces+09] in the case of T-ReX) typically provides long term planning capabilities while other components access and modify the shared plan through reading and altering the timelines of other components. The components of T-ReX (or reactors) are further hierarchically organized to ensure the reactivity of the system.

Many other techniques have been devised for the purpose of acting, notably including automata based methods [CAL92; Boh+11; Ver+05] and logic based approaches [HBL98; FL08].

Other Deliberation Functions. Other deliberation capabilities are relevant for an acting system in robotics but are beyond our interest in this chapter.

Goal reasoning is a high level capability of a system that is responsible of deciding which goals to pursue. While some planning system are capable of selecting a subset of goals in the case of an over-subscription planning problem [Smi04], the problem of goal reasoning goes beyond this capability by, for instance, synthesizing new goals to react to discrepancies (as extensively surveyed by Vattam et al. [Vat+13] and Hawes [Haw11]).

Furthermore, many robotic problems require a tight integration of task and motion planning. Indeed, in many cases, a task planner must take into account geometric constraints or the motion dynamics of a robot to solve a complex task. Many planning systems have been developed for this purpose and usually associate, at planning time, each primitive action to one possible motion for achieving it, e.g., [CAG09; CA09; Sil+14; Sri+14; Bid+15; Gar15; KL11]. Those planners further provide many search control techniques to reason on this joint search space.

5.3 General Architecture

Our system follows the very general *Actor Model* architecture [HBS73] where several components communicate and synchronize through message passing while maintaining an internal state. In this section, we give an overview of the system by presenting its main components and elements. A more detailed view of each component will be given in the subsequent sections.

5.3.1 Main Components

5.3.1.1 Platform

The platform is the component that provides a software interface to handle the various capabilities of the controlled agent. In our robotic case, the platform provides some robotic specific deliberation capabilities such as a way to compute a motion plan for grasping an object. It further provides means to actuate the robot, e.g., by executing a previously computed motion plan. It is also responsible for transforming the raw sensor output into a more intelligible form, typically through symbol anchoring.

In practice, the ROS (Robot Operating System) middleware is used to interact with the many specific tools developed by the robotics community (see Section 5.7.1).

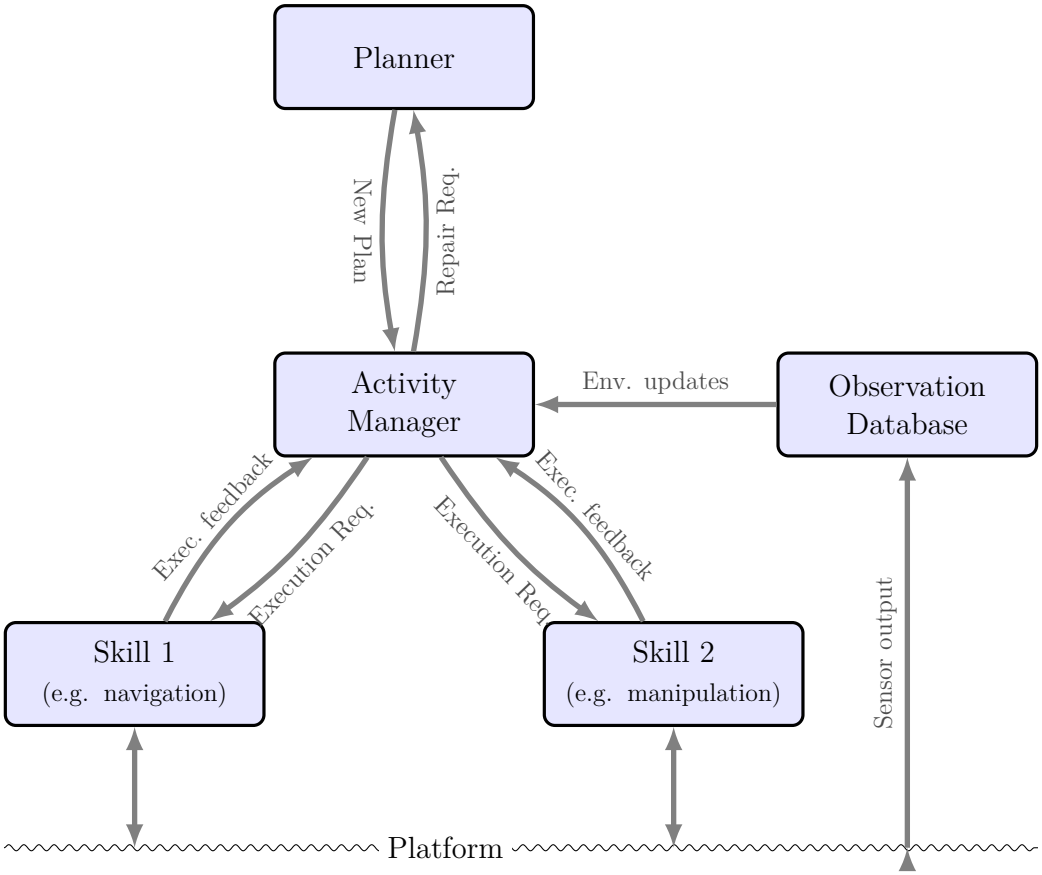


Figure 5.1: General architecture of FAPE for online planning and acting.

5.3.1.2 Skill Handlers

Skill handlers (Section 5.6) are responsible for executing a given primitive action (from the planner perspective) by refining it into one or several commands provided by the platform and overseeing their execution. A skill handler is started when an action becomes executable and terminated once it succeeds or fails.

During its execution it typically provides feedback regarding the execution status of the action (e.g. succeeded, failed) and the values of some state variables used by other components of the system.

In addition, a skill handler can refine a high-level action it is responsible for, for instance by creating new subtasks. Such new subtasks will cause the planner to extend the plan to achieve them with additional actions.

5.3.1.3 Activity manager

The activity manager (Section 5.5) is the central component of our architecture that is responsible for carrying out a plan. Given a set of goals, it maintains an executable plan that achieves them and overlooks its execution.

With the current plan, the activity manager dispatches executable actions to dedicated skill handlers. It is further responsible for integrating the feedback gathered through observations of the environment or provided by skill handlers regarding the execution of actions.

When such feedback results in an inconsistency (e.g. an action failed or an observation

does not match the predictions), it emits planning requests to either repair the current plan or get an entirely new one.

5.3.1.4 Planner

The planner is, with some minor adaptations, the planning framework we have discussed in the previous chapters of this thesis. Given a flawed plan, it is responsible in refining and extending it to make it consistent.

5.3.1.5 Observation Database

The observation database (Section 5.4) is the component responsible of maintaining the information gathered during execution. Its main objective is to map the information that was gathered through the robots sensors and make it available into anchored facts that are appropriate for usage in the rest of the system. It can further make some limited inference to extend the existing database. ORO [Lem+10] provides a good example of the desirable capabilities of this component.

5.3.2 Key Ingredients

5.3.2.1 Planning Domain

The planning domain is composed of a set of actions templates and invariants representing static knowledge on the environment (e.g. $connected(d, d')$). Actions are partitioned into executable ones, whose start timepoint is *dispatchable* and abstract ones, whose start timepoint is *structural* (see Section 4.6.4).

5.3.2.2 Time Representation

We use a discrete time representation where the set of instant is denoted by $Ticks = [\![\mathcal{O}, \mathcal{H}]\!]$, $\mathcal{O}$ being the temporal origin and $\mathcal{H}$ the temporal horizon. We refer to the current time as $t^{curr} \in Ticks$.

5.3.2.3 State Variables

The dynamics of the system is encoded with a set of state variables $\mathcal{SV}$ that either characterize the environment in which the robot evolves (e.g. the placement of a cup) or the internal state of the robot (e.g. that some sensor is currently active).

The set of state variables is not limited to the one appearing in the planning domain and can represent any value that might be usable by the activity manager or skill handlers.

5.3.2.4 Observations

An observation is a tuple (t, sv, v) denoting that a state variable $sv \in \mathcal{SV}$ was observed with the value v at time $t \in Ticks$.

We denote the set of possible observations by $\Lambda = Ticks \times \mathcal{SV} \times Val$ where Val is the set of all possible values that can be taken by state variables.

5.3.2.5 Chronicle

A chronicle remains the privileged way of representing a plan together with its associated temporal and binding constraints. The assertions of a chronicle are used to represent the expected evolution of the environment as a result of carrying out a plan.

A chronicle is extended to contain the execution status of an action that is either *pending*, *executing*, *executed* or *failed*.

5.4 Observation Database

The observation database is responsible for maintaining the knowledge on the values that has been taken by the state variables of the system.

Its maintains a database of observations $\Lambda^{DB} \subseteq \Lambda$ that is extended with new observations made by the platform or notified by other components.

In addition, it contains a set of inference functions $\mathcal{T}$. An inference function $f \in \mathcal{T}$ takes as parameters a new observation and the current database and returns a new set of facts to be added to the database.

Finally the component maintains, for each state variable sv, a list of subscribers $Subs(sv)$ that contains all components that should be notified of new observations on sv.

Messages . Synchronization with other components is done by message passing with the following messages:

- **Incoming**:
  - ***newobs***(t, sv, v): a ***newobs*** message notifies that a new fact should be added to the database, here representing that the state variable sv was observed with value v at time t. When receiving this message, the internal state is updated using the ProcessObservation procedure of Algorithm 6. New observations can be contributed by the platform (e.g. an object was observed at given location) or by skill handlers.
  - ***sub***(sv): this subscribe message is received from a component that should be notified of any new observations made on the state variable sv. The subscriber list of sv is extended to contain this component.
  - ***request***(q): where $q : \Lambda \to \Lambda$ is a function that selects a subset of the current observation database, represents a query whose result should be sent back to the original sender.
- **Outgoing**:
  - ***obs***(t, sv, v): when a new observation (t, sv, v) is added to Λ^{DB}, each component that subscribed to the new observations on sv is notified through this message.
  - ***requestanswer***(Λ_q): where Λ_q is the result of a request ***request***(q) (i.e. $\Lambda_q = q(\Lambda^{DB})$). This message is sent to the component that issued the request.

At any time, we suppose that the database does not contain any conflicting observations, i.e., that there is not two observations (t_1, sv_1, v_1) and (t_2, sv_2, v_2) such that $t_1 = t_2 \wedge sv_1 = sv_2 \wedge v_1 \neq v_2$. This requires disambiguation of observations to be made upstream, in the dedicated platform component (see Section 5.7.1).

Algorithm 6 Extends the current observation database with a new observation and the ones that can be inferred from it. Notify the other components of all new observations they have subscribed to.

function PROCESSOBSERVATION($(t, sv, v), \Lambda^{DB}$)
  $\Lambda_{inferred} \leftarrow \varnothing$
  for all $f \in \mathcal{T}$ **do** ▷ Infer new facts
    $\Lambda_{inferred} \leftarrow \Lambda_{inferred} \cup f((t, sv, v), \Lambda^{DB})$
  $\Lambda^{DB} \leftarrow \Lambda^{DB} \cup \{ (t, sv, v) \}$ ▷ Extend database with the new observation
  for all $c \in Subs(sv)$ **do** ▷ Notify all subscribers
    SEND(c, $\texttt{obs}(t, sv, v)$)
  for all $o \in \Lambda_{inferred}$ **do** ▷ Integrate and keep processing inferred facts
    $\Lambda^{DB} \leftarrow \Lambda^{DB} \cup$ PROCESSOBSERVATION(o, Λ^{DB})
  return Λ^{DB}

5.5 Activity Manager

The activity manager is the component responsible for maintaining an executable plan and overseeing its execution. Its internal state is composed of a planning domain Σ and of a chronicle ϕ representing the current plan. For each action a that appears in ϕ, we denote as $\phi_a \subseteq \phi$ the chronicle composed of all the subtasks, assertions and constraints defined in a.

The core task of the activity manager is to dispatch executable actions to their dedicated skill handler. For each currently executing action a we denote as $H(a)$ the handler that is executing it.

In addition, the activity manager serves as a central entity for incorporating plan updates (issued by skill handlers) and new observations (issued by the observation database) into the current plan. When this process results in an inconsistency or new flaws in the plan, the activity manager asks the planner to fix the current plan or provide an entirely new one.

Messages. Synchronization and exchange of information with other component is done by message passing with the following messages:

- **Incoming**:
  - $\texttt{obs}(t, sv, v)$: Notification by the observation database that a new observation has been made. The new observation is appended to a queue of pending observations Λ^{pend}.
  - $\texttt{newplan}(\phi)$: Received from the planner to notify that a new consistent plan has been found as a result of a repair or replan request. When this message is received, the current plan is replaced with ϕ.
  - $\texttt{planupdate}(\phi^{del}, \phi^{add})$: Request to alter the current plan by removing elements of ϕ^{del} and adding elements of ϕ^{add}. The request is issued by the handler $H(a)$ of an executing action a and is such that $\phi^{del} \subseteq \phi_a$ (i.e. the handler can only request the removal of statements in the action it is responsible for). A plan update can also mark an action as *failed* or *succeeded*. When received, a plan update is appended to a queue U^{pend} of pending plan updates.

- ***newgoal***(g): Notification that a new goal should be pursued. The new goal is appended to a queue G^{pend} of pending goals.

- **Outgoing**:
  - ***exec***(a): this message is sent to a skill handler to notify it that is should carry out the execution of an action a.
  - ***active***$(tp, [earliest, latest])$: notification that a *dispatchable* timepoint tp can be executed in the temporal window $[earliest, latest]$. This message is sent to the skill handler of the action a in which tp was defined (i.e. $tp \in \phi_a$).
  - ***repair***(ϕ'): Sent to the planner to request that a flawed plan ϕ' be repaired.
  - ***sub***(sv): Request sent to the observation database to be notified of any new observation made on the state variable sv. The activity manager subscribes to all state variables that appear in the planning domain Σ.

5.5.1 Main Acting Loop

The core of the activity manager is composed of the ACT procedure (Algorithm 7) that is executed every time a *tick* occurs or a ***newplan***(ϕ) message is received.

In between two ACT invocations, the activity manager keeps track of all the request it has received regarding new goals, new plan update requests and new observations it received. Those are respectively stored in G^{pend}, U^{pend} and Λ^{pend}.

The ACT procedure starts by integrating into ϕ all pending goals, updates and observations (lines 1–7). The MERGEOBS procedure, detailed in Section 5.5.2, essentially identifies the events associated to the new observations and propagate temporal constraints to maintain the plan consistent. When an observation cannot be reconciled with existing assertions in the plan, it results in a conflict making the current plan inconsistent.

When the integration of pending updates results in a valid plan (i.e. there was no conflicting observations and the integration of goals and updates did not introduce any flaw), the executable actions in the plan are dispatched by the DISPATCH procedure (detailed in Section 5.5.3). Dispatching essentially consists in starting new skill handlers to carry out the execution of actions that can be started at the current tick t^{curr}.

Algorithm 7 ACT merges observations and plan updates into the current plan and dispatches the resulting plan if it is consistent. Otherwise, a repair or replan operation is carried out.

```
1:  procedure ACT(φ, G^pend, U^pend, Λ^pend)
2:      for all g ∈ G^pend do                          ▷ Integrate new goals
3:          φ ← φ ∪ g
4:      for all (φ^del, φ^add) ∈ U^pend do             ▷ Integrate plan updates
5:          φ ← φ \ φ^del ∪ φ^add
6:      for all λ ∈ Λ^pend do          ▷ Merge observations (including inferred ones)
7:          φ ← MERGEOBS(φ, λ)
8:      if φ is a solution plan then
9:          DISPATCH(φ)
10:     else
11:         REPAIRORREPLAN(φ)
```

If the integration steps resulted in an inconsistency, the activity manager will try to repair the plan. This procedure, detailed in Section 5.5.4, typically consists in removing some (when repairing) or all (when replanning) of the pending actions in ϕ and issuing a plan request to the planner. When receiving the new plan from the planner, the activity manager will restart a new ACT cycle.

5.5.2 Observation Merging

Observation merging consists in aligning the predictions associated to the current plan with the observations that have been made so far. A given observation (t, sv, v), where sv is a state variable appearing in ϕ, must be placed on the timeline giving the expected evolution of the state variable sv. This process is made complex by the fact that a timeline only gives a partial view of the evolution of a state variable. The value of a state variable sv might be undefined at a time t when no assertion was made on its value at an earlier time and or if t overlaps a transition assertion. Furthermore, the controllable timepoints associated with a timeline can retain some flexibility and some timepoints can be contingent, meaning that their occurrence time is a priori unknown. When a plan contains contingent events, an observation further allows to infer knowledge on whether or not they have already occurred.

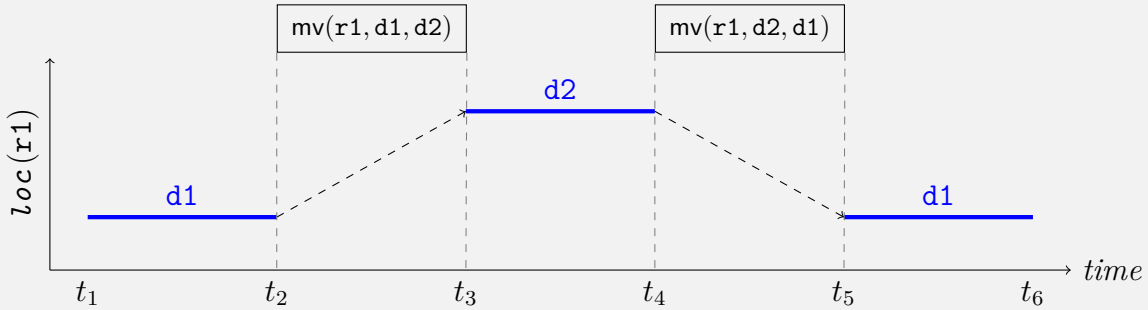
Given an observation (t, sv, v), we are first interested in finding a temporal interval $[t_s, t_e]$ where the observation is consistent with the predicted values of sv.

Definition 5.5.1 (Merge Candidate)**.** Given a chronicle ϕ, a temporal interval $[t_s, t_e]$ is a merge candidate for an observation (t, sv, v) iff:

- $t \in [t_s, t_e]$ is consistent with $\mathcal{C}_\phi$,
- $\forall t' \in [t_s, t_e]$, $\langle sv{=}v \rangle$ or $\langle sv{=}\text{UNDEFINED} \rangle$ holds at time t',
- neither $\langle sv{=}v \rangle$ nor $\langle sv{=}\text{UNDEFINED} \rangle$ holds at times $t_s - 1$ and $t_e + 1$.

Example 5.1. Consider the timeline below that denotes the evolution of the state variable $\texttt{loc}(\texttt{r1})$ as a result of two move actions. Assuming that the only temporal constraints on the timepoints $t_{1..6}$ is a total order, the merge candidate for an observation $(t, \texttt{loc}(\texttt{r1}), \texttt{d2})$ is the temporal interval $]t_2, t_5[$. Indeed in this interval the value of $\texttt{loc}(\texttt{r1})$ is either undefined (on $]t_2, t_3[\cup]t_4, t_5[$) or equal to d2 (on $[t_3, t_4]$).

An observation $(t, \texttt{loc}(\texttt{r1}), \texttt{d1})$ would have two merge candidates $[\mathcal{O}, t_3[$ and $]t_4, \mathcal{H}]$, $\mathcal{O}$ and $\mathcal{H}$ denoting the time origin and temporal horizon respectively.



Perhaps surprisingly, an observation $(t, \texttt{loc}(\texttt{r1}), \texttt{d4})$ would also have two merge

candidates $]t_2, t_3[$ and $]t_4, t_5[$ in this example. Indeed, these intervals represent durative changes during which the value of $\texttt{loc}(\texttt{r1})$ is not defined and could take any value. If this is problematic (e.g. the agent should not pass through in any intermediate location), the action model could be adapted to encode this fact. For instance, one could represent the effect of moving from one location to another with an instantaneous transition to occur at a contingent timepoint t during the move action.

Let us first assume that each observation has at most one merge candidate. This is often verified in practice as observations are only made for past or current timepoints and the executed part of the plan typically has no temporal flexibility left. In our example above, where the timeline denotes the changes due to two sequential move actions, the occurrence time of all time points before t^{curr} would be fixed. These additional constraints would result in a single merge candidate for $(t, \texttt{loc}(\texttt{r1}), \texttt{d1})$.

In the exceptional case where multiple merge candidates are possible, we would greedily select the earliest merge candidate. This can be problematic in cases where the contingent evolution of a state variable is not entirely observable and the contingent durations allow for indistinguishable scenarios given a limited set of observations. This is however not a case we have encountered in practice. In fact, the techniques developed in Chapter 4 allow to enforce sufficient observation in cases where the occurrence time of contingent events matter.

Definition 5.5.2. An assertion $\alpha \in \mathcal{F}_\phi$ is necessarily conflicting with an observation (t, sv, v) iff *(i)* α is an assertion on sv, *(ii)* t necessarily overlaps α (i.e. $\mathcal{C}_\phi \models t \in [start(\alpha), end(\alpha)]$), and *(iii)* α asserts a value incompatible with v.

Definition 5.5.3. An assertion $\alpha \in \mathcal{F}_\phi$ possibly overlaps with an observation (t, sv, v) iff α is *(i)* an assertion on sv and *(ii)* t possibly overlaps α (i.e. $t \in [start(\alpha), end(\alpha)]$ is consistent with $\mathcal{C}_\phi$).

The procedure for merging an observation $\lambda = (t, sv, v)$ into the current plan is given in Algorithm 8. The procedure starts by finding a merge candidate $[t_s, t_e]$ for λ. If such a candidate exists it augments ϕ with temporal constraints to enforce that t be in $[t_s, t_e]$. In the case where t_s denotes a contingent event, it is marked as *executed* and instantiated to the latest time before t compatible with its bounds.[1] In the case where t_e denotes a contingent event, the bounds on its incoming contingent link are updated to reflect the fact that it cannot occur before time t.

If there is no assertion possibly overlapping λ, then a new a priori supported assertion is added to the plan. This case typically occurs when the initial problem (corresponding to an empty plan) is constructed by merging all observations in it.

If there was no merge candidate for this observation, then all conflicting assertions are marked as such in the plan. This makes the plan inconsistent and will be handled by repairing the plan.

5.5.3 Dispatching

The execution of a plan relies on the dispatching algorithms for STNU. More specifically, a plan ϕ is associated with a POSTNU that is compiled to an STNU through our method

[1] By an abuse of notation and in accordance with the literature (e.g. [NKD14a]), we say that a contingent timepoint is *executed* once it has been observed.

Algorithm 8 Extends the current plan ϕ to account an observation $\lambda = (t, sv, v)$. It further identifies assertions that are conflicting with the observation.

procedure MERGEOBS(ϕ, λ)
  if $\exists$ earliest merge candidate $[t_s, t_e]$ for λ **then**
    $\mathcal{C}_\phi \leftarrow \mathcal{C}_\phi \cup \{\, t \in [t_s, t_e] \,\}$
    if t_s non-executed contingent **then**
      INSTANTIATE(ϕ, t_s, t)
    if t_e contingent timepoint **then**
      DELAYPOSSIBLEOCCURRENCE(ϕ, t_e, t)
    if $\nexists\ \alpha \in \mathcal{F}_\phi$ possibly overlapping with λ **then**
      $\mathcal{F}_\phi \leftarrow \mathcal{F}_\phi \cup \{\, [t]\, sv{:=}v \,\}$ ▷ Add an a priori supported assertion
  else
    for all $\alpha \in \mathcal{F}_\phi$ conflicting with λ **do**
      $\phi \leftarrow$ MARKCONFLICTING(ϕ, α, λ)
  return ϕ

detailed in Section 4.4. This STNU is maintained in a dispatchable form by using the algorithm of Morris [Mor14] that infers all edges needed for dispatching the STNU in addition to those needed for checking its dynamic controllability.

We say that a dispatchable timepoint *tp* is *enabled* if $tp = t^{curr}$ is a consistent assignment for *tp*. In a dispatchable STNU, it essentially means that *(i)* all dispatchable timepoints before it are *executed*, *(ii)* all its wait constraints on contingent events are verified and *(iii)* the current time t^{curr} is in the allowed execution window of *tp*.

Our DISPATCH procedure (Algorithm 9) takes as parameter a dispatchable plan ϕ. For any timepoint that is *enabled* at the current time and corresponds to the start timepoint of an action a, a new skill handler is started to carry out the execution of a. The skill handler is notified of the action to execute and *tp* is marked as executed.

Beside its start timepoint, an action may contain other dispatchable timepoints. Such timepoints can be used to control the start time of subactivities while obeying synchronization constraints with other events. For instance, in a Charge action that requires plugging in and unplugging the robot, one must take a decision regarding when to start unplugging. For such timepoints, our DISPATCH procedure does not take the responsibility of instantiating the timepoint. Instead, the handler of the action is notified that one of the timepoints it is responsible for is *enabled* and gives the allowed execution window $[earliest, latest]$ where *earliest* and *latest* respectively denote the earliest and latest execution time of *tp* (i.e. $dist_{STN}(\mathcal{O}, tp)$ and $-dist_{STN}(tp, \mathcal{O})$). The handler will decide when to execute it and report on its instantiation to the activity manager through a plan update.

Last, if there is an *enabled* timepoint that is not attached to any action, then this timepoint is instantiated.

5.5.4 Repair and Replan Requests

The last responsibility of the activity manager is to maintain the plan consistent by issuing repair or replanning requests. Indeed, merging observations might result in conflicts with existing assertions. Likewise, the plan updates sent by skill handlers can reveal an action failure or introduce new flaws. These scenarios would result in a plan that is not

Algorithm 9 Dispatching algorithm

```
H: mapping from each executing action to its skill handler
procedure DISPATCH(φ)
    enabled ← dispatchable timepoints in C_φ instantiable at t^curr
    for all tp ∈ enabled do
        if tp is the start timepoint of an action a ∈ π_φ then
            H(a) ← new skill handler for a
            INSTANTIATE(φ, tp, t^curr)
            SEND(H(a), exec(a))
            MARKEXECUTING(φ, a)
        else if tp is a controllable timepoint inside an action a then
            SEND(H(a), enabled(tp), [earliest, latest])
        else
            INSTANTIATE(φ, tp, t^curr)
```

dispatchable because of its inconsistencies. FAPE handles such inconsistencies by first attempting to repair the plan. This process is done by removing problematic actions that are not executing and handing out the restricted plan to the planner.

5.5.4.1 Plan Repair

Given an inconsistent plan ϕ, the repair procedure builds a chronicle ϕ^{repair} by:

- removing from ϕ any action that was marked as failed;
- for any assertion α marked as conflicting in ϕ, if α was defined in an action instance a, then a is removed from ϕ. Otherwise, α alone is removed. The observation that was conflicting with α is introduced in the chronicle as an *a priori* supported assertion.

A notable exception is that no action currently executing is ever removed, even if its assertions are conflicting with an observation. In this case, the responsibility of declaring the action as failed is left to the skill handler. This condition ensures that for any executing action a all its elements will stay in the current plan of the activity manager (i.e. $\phi_a \subseteq \phi$). The skill handler is responsible for maintaining the model of the action consistent (e.g. by updating the temporal scope of the action if its execution takes longer than initially accounted for in the model) or declaring it failed.

Action Removal. The removal of an action a from a plan ϕ is done as follows:

- all assertions, constraints, tasks, timepoints and variables introduced by the action are removed from ϕ (i.e. $\phi \leftarrow \phi \setminus \phi_a$);
- if the action had any subtasks, then all pending actions refining one of those subtasks are recursively removed as well;
- any plan refinement involving an element of a (i.e. its timepoints, object variables, assertions and subtasks) is removed. For instance, any causal link that supports or is supported by an assertion of a is removed as well as any separation constraint involving a temporal or object variable of a.

5.5.4.2 Repairing vs Replanning

Plan repair is typically attempted first with a domain-dependent timeout. If the planner fails to find a plan in this time window, the activity manager will start a replanning phase. Replanning is a special case of plan repair where all pending actions are removed from the plan.

Several variations of this scheme are possible. If the initial repair request failed, one could for instance try to remove more actions (e.g. those with a broken incoming causal link or whose subactions were removed), until the planner can come up with a valid plan.

Another variation would to be to initialize the search space of the planner with multiple partial plans, each one with a different set of removed actions. The planner would then choose the most promising partial plan to expand based on its own heuristics.

5.5.4.3 Dealing with Orphan Task-Dependent Actions

The action removals made for the purpose of plan repair or replanning can result in *orphan* actions: task-dependent actions that do not refine any task. For instance, if an executing action a was refining the subtask of a removed action, then a would still be in the plan but the task it refined would be absent from ϕ^{repair}.

We consider two ways of handling such orphan actions. The first one is simply to ignore them and allow the plan to have orphan actions. This is a convenient choice as it allows to keep the planning domain and the planner's implementation untouched. However it results in an unsoundness as FAPE could carry out the execution of a plan where some actions do not obey the hierarchical constraints of the domain.

A more involved possibility is to ensure that, after repairing, the solution plan returned by the planner does not contain any orphan action. We now detail how this can be achieved through an additional flaw definition that handles orphan actions.

Definition 5.5.4 (Orphan Action Flaw)**.** Any task-dependent action that does not refine any task constitutes an *orphan action flaw*.

Dealing with orphan actions is done in a fashion similar to the one for handling unsupported assertions. Instead of looking for an action that can provide an effect for causal support, we look for an action that can provide a task to be refined by the orphan action. We briefly sketch here the resolvers for an orphan action. The specifics can be obtained by simply adapting the resolvers of unsupported assertions to subtasks instead of effects (Section 3.2.2.1).

Definition 5.5.5 (Possible Subtasks)**.** An action a has a set of possible subtasks defined recursively as its own subtasks and the possible subtasks of all actions that can refine one of its subtasks. The possible subtasks of a task is the union of the possible subtasks of all actions that can refine it.

Resolvers for Orphan Actions. Like for unsupported assertions, orphan actions are handled by either selecting an existing task to refine or selecting a source from which the task to refine will be taken. In the second case, the actual resolution of the flaw is delayed. More specifically, given an orphan action a in a chronicle ϕ, we distinguish three types of resolvers:

- **Direct refinement:** an orphan action flaw can be resolved by making the orphan action a refine an unrefined task $\tau \in \pi_\phi$ currently in the plan.
- **Delayed refinement from existing task**: given a task $\tau \in \pi_\phi$ such that $task(a) \in possible\_subtasks(\tau)$, the planner can commit to choose the task refined by a in the tasks inserted while decomposing τ. In this case, the resolution of the orphan action flaw is delayed until τ has been refined by an other action.
- **Delayed refinement from new action**: given a *free* action template B such that $task(a) \in possible\_subtasks(B)$ a possible resolver is to *(i)* insert a new instance b of B, and *(ii)* commit to choose the task refined by a in the direct or indirect subtasks of b.

For maintaining completeness of the planner, an unrefined task flaw has another type of resolver that involves marking an orphan action as refining that task.

Explicitly handling orphan actions allows enforcing the soundness of the system: regardless of the failure that occurs during execution: any task-dependent action will always be considered as part of an higher level action and will thus respect all hierarchical constraints defined in it. On the down side, it places much more stress on the domain designer as one must now ensure that the hierarchical domain is robust enough to handle various cases of plan failures.

The choice of ignoring orphan actions or not is inherently domain dependent and especially depends on whether task-dependent actions are used to improve the planner's efficiency or to enforce necessary constraints between actions. The planner implementation supports both possibilities.

5.6 Skill Handlers

A skill handler is a component responsible for carrying out the execution of a single action instance. Executing an action can be done by two main mechanisms. One can send commands to the platform that will result in altering the environment. For instance, one could trigger a ROS command whose effect is to make the robot navigate to a given location. Another way to perform an action is to extend it with new subtasks. These will trigger a plan repair in which the planner will add new actions to the current plan.

In either case, a skill handler is responsible for monitoring the execution of the action (e.g. by controlling that the observations match its expectations) and of reporting on its execution. The latter is done by sending plan updates to the activity manager that reflect the current status of the action (e.g. the action failed or succeeded) and that adapts the current model of the action (e.g. the action has new subtask or one of its timepoints was instantiated). A skill handler can also send new observations to the observation database that reflect its activity or the evolution of the environment. For instance, an action for scanning a surface would report that a given table `T` has been scanned at a given time t by providing a new observation $(t, scanned(\texttt{T}), \top)$.

As for other components of the system, a skill handler communicates with other components through message passing. We consider the following messages to be received and sent by a skill handler:

- **Incoming**:

  - ***exec***(a): the first message that is sent by the activity manager to a skill handler. It requests the execution of an action a. We denote as ϕ_a the chronicle composed of all subtasks, assertions and constraints defined in a.
  - ***obs***(t, sv, v): notification sent by the observation database that a new observation has been made on the state variable sv.
  - ***active***$(tp, [earliest, latest])$: notification from the activity manager that a dispatchable time points $tp \in \phi_a$ is enabled and can be dispatched within the execution window $[earliest, latest]$.

- **Outgoing**:

  - ***newobs***(t, sv, v): notification sent to the observation database that it should record this observation and notify any component that subscribed to the observations of the state variable sv.
  - ***request***(q): request sent to the observation database to get any past observation matching the query q.
  - ***sub***(sv): request sent to the observation database to be notified of any new observation made on the state variable sv.
  - ***planupdate***(ϕ^{del}, ϕ^{add}): where $\phi^{del} \subseteq \phi_a$, is a request sent to the activity manager to update the model of action a. The current model is adapted as follows: $\phi_a \leftarrow \phi_a \setminus \phi^{del} \cup \phi^{add}$. In addition to the usual definition of a chronicle, ϕ^{add} can also mark an action as *failed* or *executed*.

This general scheme allows for quite general capabilities: a skill handler can freely update the model of the action it is responsible for. Let us now detail the two cases we are interested in that respectively handle the execution of a primitive action and of a high level action.

5.6.1 Execution of a Primitive Action

As an example of a primitive action, we consider a $\mathsf{Go}(r, l)$ action where a robot r will navigate to location l. An ANML model of this Go action is given in Appendix B.5. The action has two dispatchable timepoints: its start timepoint at which the robot must plan a trajectory to its target location and a t_{exec_traj} timepoint where the robot will start navigating to its target location by executing the previously computed trajectory. Its trajectory planning activity takes an uncontrollable duration and requires the path planner to be available (i.e. it should have no concurrent requests). Its navigation activity also has a (much larger) uncontrollable duration and will end once the robot reaches its target location. For the first phase of this action (i.e. until t_{exec_charge}) no requirement is made on the actual location of the robot.

The skill handler for the Go action follows an automaton whose simplified version is given in Figure 5.2. Starting from an *Init* state, the handler awaits for an ***exec***$(\mathsf{Go}(r, l))$ message. When it is received, the handler will subscribe to any new observation made on the state variable $loc(r)$ that keeps track of the current location of the robot. It then starts planning its trajectory by issuing commands to the appropriate component of the platform in order to find a precise and reachable target location adapted to the current task. For instance if the Go action is part of a high-level action to pick an object, the skill handler looks for a final location close enough to the object to be able to pick it,

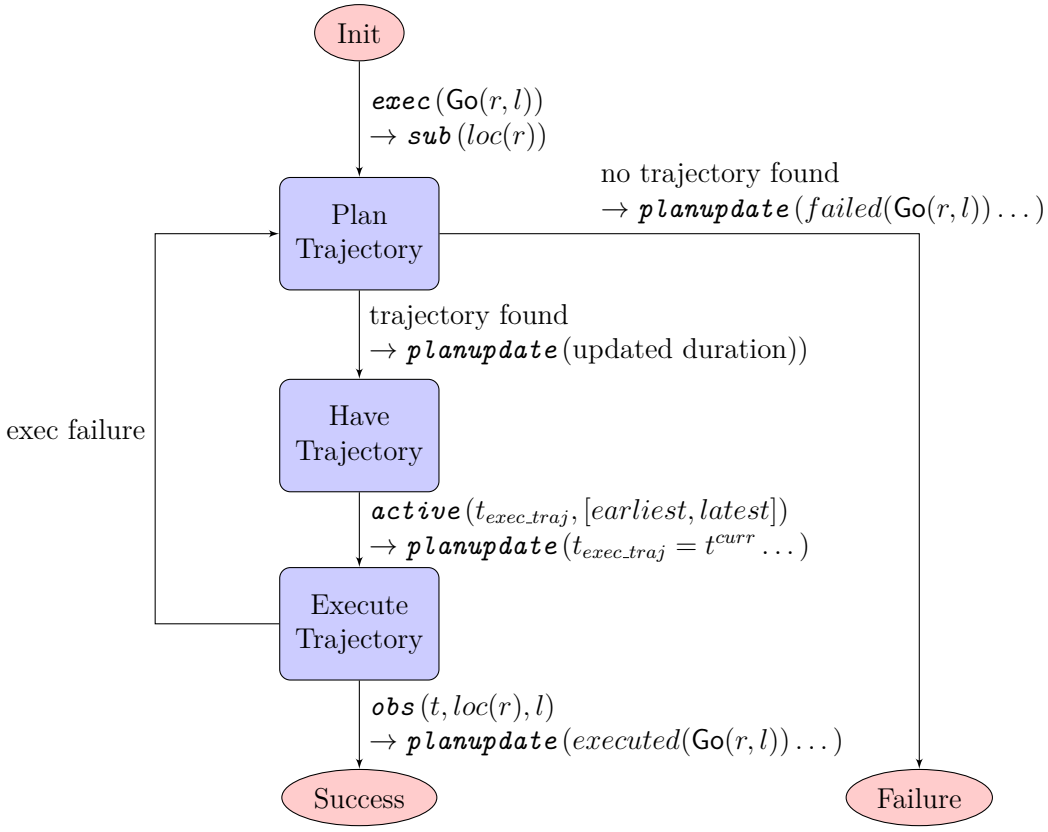


Figure 5.2: Simplified workflow of the Go action for bringing a robot r to a location l.

using inverse kinematics in a collision aware motion planner such as GTP (Section 5.7.1). If no target pose and trajectory is found, then the action is considered as failed and the activity manager is notified of the failure with a plan update message. Otherwise, the skill handler moves to a *Have Trajectory* state. A plan update is sent so that the duration of the navigation activity reflects the actual duration of executing the now available trajectory (before that, its duration was encoded with conservative lower and upper bounds).

The skill handler will stay in the *Have Trajectory* state until it receives a notification that his t_{exec_traj} timepoint is enabled, i.e., that all temporal requirements to start navigating are met. At this point, the robot will start executing the trajectory through the dedicated commands of the platform (e.g. the MoveBase node of a ROS-operated robot). If it reaches its target, the skill handler notifies the activity manager of the success and exits. Otherwise, if the navigation failed (e.g. the robot faced an obstacle that was not on its map), the skill handler will go back to planning a trajectory.[2]

This skill handler illustrates some of the key capabilities needed for acting. Indeed, even in the case of simple navigation action, one must first refine the planner's primitive action into a detailed trajectory. When possible, this should be done ahead of time so that the execution faces no unneeded latency. Furthermore, this refinement phase lifts some uncertainty regarding the actual behavior of the planner's primitive, here by providing an updated information regarding its duration. Acting further requires to monitor the

[2]The use of SLAM techniques typically allow a robot to continuously update its map of the environment to account for newly detected obstacles when computing a new trajectory.

execution of the action and react to failures.

While, in this example, each activity is started as soon as possible, this not necessary the best route. Consider for instance an activity to unplug a robot that is currently charging. The fact that the initial timepoint of this activity becomes *enabled* denotes the fact that the robot now has just enough battery to carry out the rest of the plan (assuming that everything goes according to the plan). Opposed to unplugging as soon as possible, an often wiser strategy would be to wait until the battery is fully charged or the robot is needed for another activity.

A skill handler is thus the place where all deliberations that are beyond the scope of task planning are made at acting time. This approach assumes a downward refinement property between the high level representation used by the planner and the low level primitive commands of the platform. Essentially, this means that we assume that the (high level) effects of an action can be achieved by a skill handler, possibly involving various local recovery behavior.

5.6.2 Online Refinement of High Level Actions

In some cases, it is either impractical or impossible to come up with a complete high level plan, for instance because some information might be missing. Sometimes, it is also desirable for the planner to come up with an abstract plan whose details will be found later. For instance, in the process of planning a trip to a remote country in a few months, I am mostly interested in the flights I need to book. However, the details of my activities at the airport do not need to be known until the last minute.

In this section, we present a way to perform online refinement of high level actions. A high level action provides a conservative model of its requirements and of its effects. The action model will be adapted online to provide new subtasks needed to achieve its intended behavior.

For this purpose, we take the example of an action for transporting an object to a given location. Since we don't assume a complete knowledge of the environment, the initial location of the object might be unknown. Achieving a transport task in these conditions first requires looking at the possible locations of the object until it is found. It is only when found that details of the rest of the plan will be known (e.g. how and where to pick the object). One possibility in this scenario would be to build a contingent plan that is conditioned on all possible locations of the object. This would result in a highly combinatorial process. Instead, we use a more tractable approach of having a high level SearchTransport action whose skill handler will decide online which location to scan next and add new subtasks that will be refined through a planning process. Once the object is found, the handler will issue a new transport task that will be refined by the planner with Pick and Place primitive actions.

The ANML model of the SearchTransport action is given in Appendix B.5. SearchTransport(r, i, l) requires a robot r to transport an item i to a location l. The initial model of this action is conservative: *(i)* it takes a potentially long unknown duration, and *(ii)* it requires that both a gripper and the head of r be usable for the duration of the action. It furthermore states that the location of r is undefined for the duration of the action but that it will end up in a location from which the surface l can be reached. Finally, it states that the item i will be in l at the end of the action. This model essentially state the sufficient resources to achieve the high level goals.

The behavior of the skill handler is given in the (simplified) flow chart of Figure 5.3.

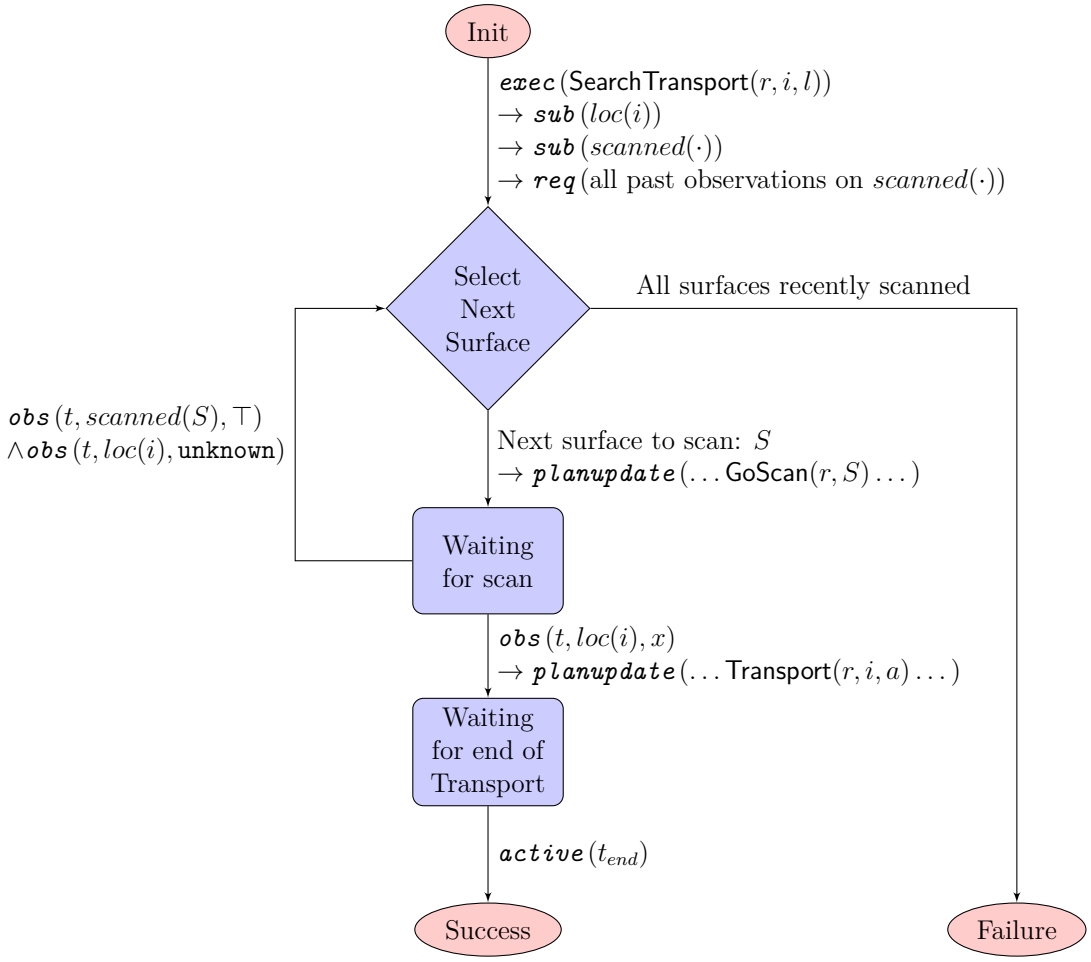


Figure 5.3: Simplified flow chart of the skill handler of SearchTransport.

Starting from an *Init* state, it will first require all new observations on the location of the object as well as all (past and future) observations regarding which surface has been scanned.

Given this knowledge, it will then decide which location to look for item i next. This decision is made by selecting the closest surface S that has not been scanned recently.[3] If there is no such surface, then the action is considered as failed. Otherwise, the action model is adapted to contain a new GoScan(r, S) subtask. The other assertions regarding the location of the robot and the status of its gripper and head are delayed to after the realization of the GoScan task. This is achieved by issuing a **planupdate**(ϕ^{del}, ϕ^{add}) whose elements are given below (for simplicity we omitted the assertions regarding the

[3]Such a decision should ideally be done in more high-level knowledge-based module (e.g. ORO[Lem+10]) that is not available in the current architecture.

gripper and the head):

$$\begin{aligned}\phi^{del} = \{&\varnothing,\\ &\{\,[t_{start}, t_{end}]\, loc(r){:}\, r\_start \mapsto r\_end, \dots\}\\ &\{\,t_{start} \xrightarrow{[30,300]} t_{end}, \dots\}\\ &\}\end{aligned}$$

$$\begin{aligned}\phi^{add} = \{&\{\,[t_s, t_e]\, GoScan(r, S)\,\}\\ &\{\,[t_e, t_{end}]\, loc(r){:}\, r\_start2 \mapsto r\_end, \dots\}\\ &\{\,t_e \xrightarrow{[20,280]} t_{end}, \dots\}\\ &\}\end{aligned}$$

This plan update will result in a new *unrefined task* flaw in the current plan, and the plan will be extended to refine this task with additional actions. It should be noted that such a repair mechanism is local as newly inserted actions will take the place of the original assertions that were removed.

A view of the updated SearchTransport action is given in Figure 5.4 after *(i)* deciding to first scan a surface `s1`, *(ii)* having repaired the plan to refine the resulting GoScan$(r, \texttt{s1})$ task, and *(iii)* having already executed the first Go action.

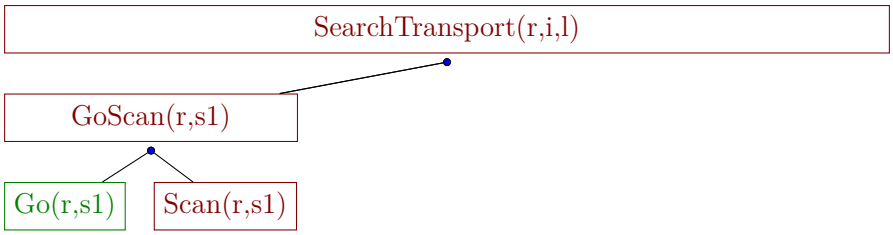


Figure 5.4: Refinement of SearchTransport while scanning a first surface `s1`. In green are the actions already executed and in red are the actions currently executing.

This process of selecting a new surface to scan and adding a new GoScan subtask continues until the location of item i becomes known (i.e. an $\textit{obs}\,(t, loc(i), x)$ message is received). At this point, the handler will add a last Transport subtask and exit once it has been carried out.

An a posteriori view of the decomposition of SearchTransport after having scanned a first surface `s1` and scanned a second surface `s2` where the object was identified is given in Figure 5.5.

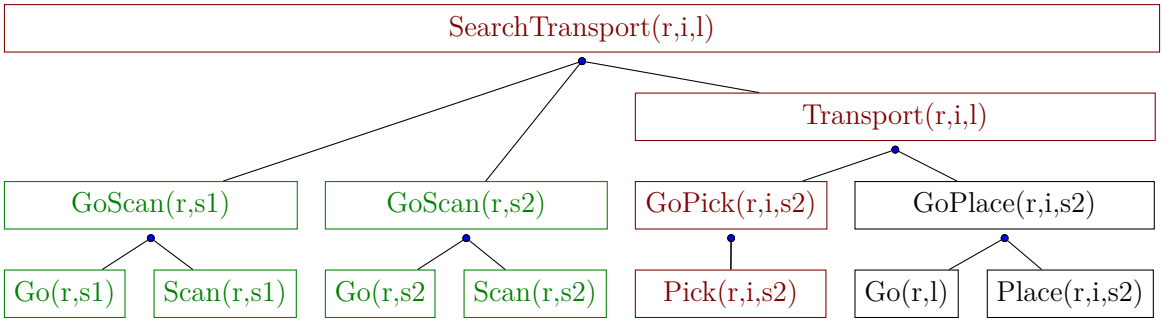


Figure 5.5: Refinement of SearchTransport after the scan of surface s2 revealed the position of i. In green are the action already executed, in red are the one currently executing and in black are the pending ones.

5.7 Experimental Results

5.7.1 Platform Settings

The platform is built on top of the ROS (Robot Operating System) middleware, using the Gazebo simulator to provide a physics engine for robotic simulation. The simulated environment contains three tables with colored tapes serving as manipulable objects (Figure 5.6). A single PR2 robot is simulated and the various components needed for controlling it are made available through standard ROS interfaces (e.g. controllers and joint positions).

Localization and mapping are performed with the ROS pr2_2dnav package that provides interfaces for localizing the robot in its environment through SLAM techniques as well as primitives for sending the robot to given coordinates.

The software Toaster[4] provides a central point for gathering observation from multiple sensors. In our setting, it uses the position of objects from the internal model of gazebo, thus having complete and exact knowledge of their poses. The position of the robot is the one computed by pr2_2dnav using SLAM. This position estimation typically differs from the actual location by a few centimeters, with this error being passed to the relative positions of objects relatively to the robot.

GTP [WGA16] provides motion planning capabilities for the PR2 robot. Most interestingly for us is its capability to synchronize its internal model with the one advertised by Toaster and the capability of computing poses and trajectories given object names (rather than fully defined joint positions). It is used for finding collision free target poses and computing trajectories for object manipulation, e.g., for picking up an object.

PR2Motion[5] provides many ROS actions for actuating a PR2 robot. It is used for controlling the head and torso poses as well as executing the more involved trajectories computed by GTP.

5.7.2 Experimental Setting and Results

The system is used for moving objects between tables using the action models previously described in this chapter. It initially starts with no knowledge about the location of objects and is given one or multiple tasks of searching and transporting objects.

[4] https://github.com/laas/toaster
[5] https://git.openrobots.org/projects/pr2motion-genom3

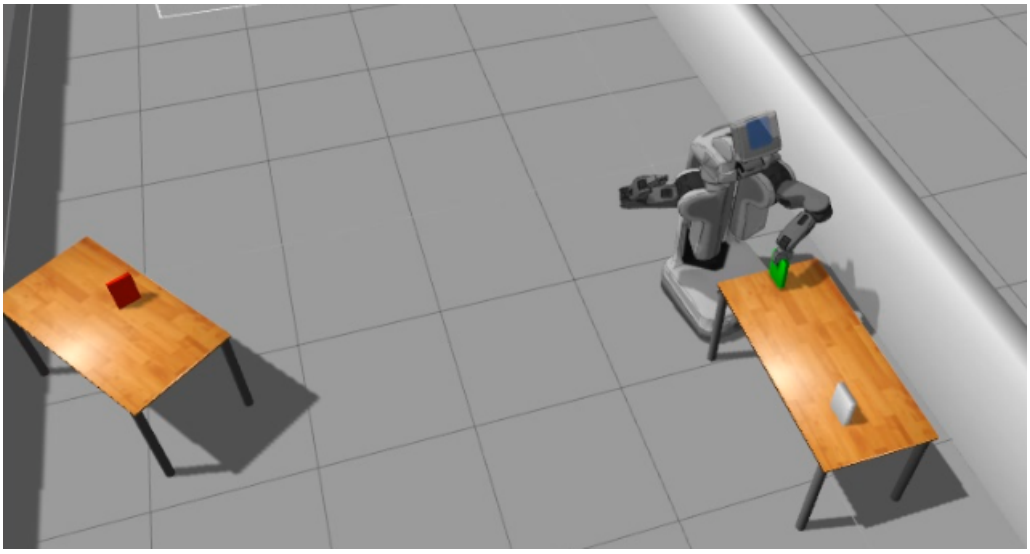


Figure 5.6: PR2 robot executing a plan in the Gazebo simluator, with two of the three tables being visible.

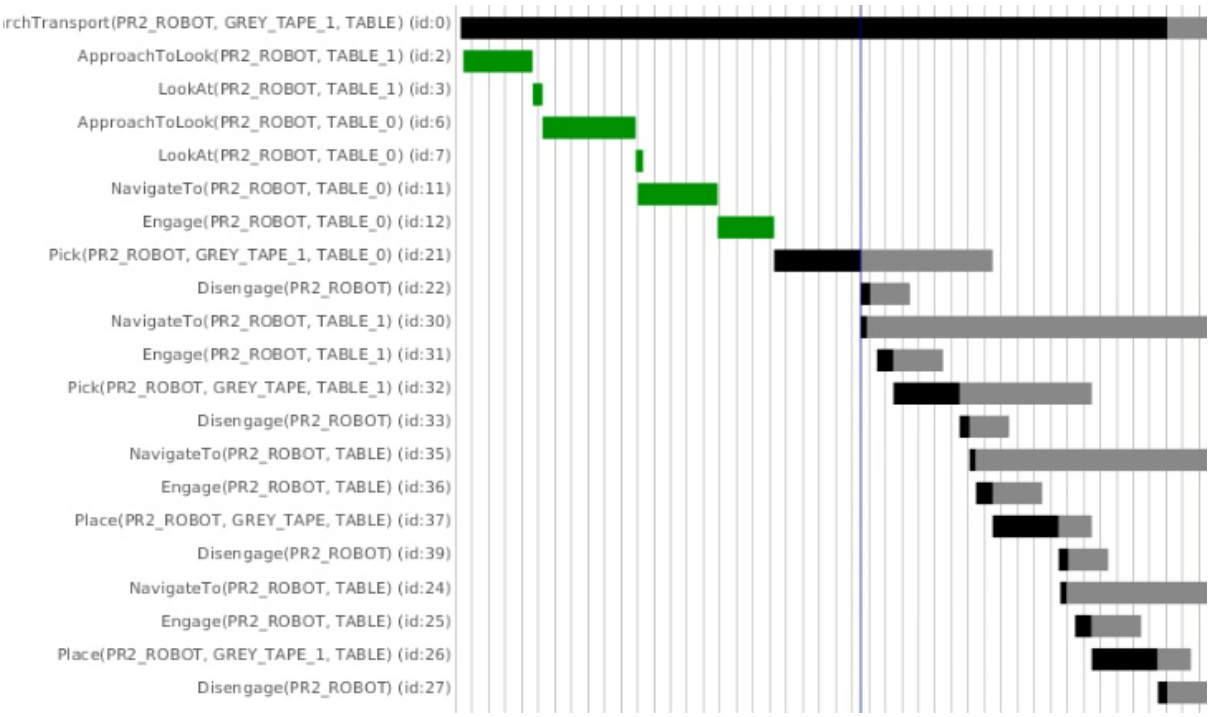


Figure 5.7: View of the current plan of the PR2 when performing two transportation tasks.

None of the positions in the environment are precomputed and skill handlers for navigation rely on GTP for testing the feasibility of various target poses adapted to the task at hand, e.g. close enough of an object to allow picking it up while ensuring that the robot does not collide with the table. The skill handler for picking up objects also relies on the GTP motion planner for computing the arm trajectories. Said trajectories are sent to PR2Motion for execution with additional commands actuating the gripper. Due to incompatibilities between the various pieces of software used, the place action is performed by simply releasing the object above the table.

The observation database is updated by retrieving the pose of the various components in the environment from Toaster. This information is filtered when made available to FAPE to only include the position of objects that are currently visible by the robot (i.e. the robot looks in their direction and is close by).

The planner is used with no tuning of its default heuristics. When given a planning request, the planner is first run without checking dynamic controllability during search. If the plan found is not DC (e.g. a deadline cannot be guaranteed to be met given the temporal uncertainty), then search is restarted with dynamic controllability checking of every partial plan. When facing an inconsistent plan, repair is first tried with a timeout of 200 milliseconds. If no valid plan is found in this time window, replanning is triggered, only keeping actions that were previously executed and the ones currently executing. During a planning phase, the activity manager stays active to monitor currently executing actions but will not dispatch any new action.

All components of the system are allowed to run concurrently and communicate through message passing. The activity manager is run at a frequency of 20Hz at which it integrates all observations, dispatches actions and sends planning request to the planner.

We evaluated the system on 5 configurations, each involving 4 objects and three tables. In each setting, FAPE was given an initial goal to achieve in the form of a transportation task and two additional goals which were added online. Each objective was associated with a deadline. In each case, the system was able provide a full plan for the tasks at hand or prove some of the tasks to be unfeasible and provide a plan for the remaining ones. The entire plans included between 18 and 26 primitive actions and their executions required from 3 to 6 minutes.

Two types of action failure occurred in the context of the pick action. The first one was to find an object to be unreachable by the robot as notified by the motion planner. The second one was to find out that the object was not actually picked after executing the pick trajectory, which can be detected by monitoring the position of the gripper. In the former case, the action was declared as failed and the object as unreachable, causing the system to abandon the unfeasible task after replanning. In the latter case, the skill handler would try a second pick before declaring the object as unreachable.

Table 5.1 provides the average and maximal runtimes for various operations involving the planner. The relatively small difference between between plan repair and replanning is mostly explained by the relatively short plans at hand and the presence of a single agent, that make most of the actions interdependent. The need for Dynamic Controllability arises when a deadline might be violated given the uncertain duration of actions. Its activation significantly increases the cost of planning but allows the planner to find dynamically controllable plans in temporally constrained settings. Last, the planner was able to rule out infeasible plans very quickly, typically at the time of the initial reachability analysis. This capability was critical to allow the planner to quickly find a possible combination of goals when facing overconstrained problems.

	Average runtime (ms)	**Max runtime (ms)**
Plan repair	96	200 (timeout)
(Re)Planning (DC not needed)	196	703
(Re)Planning (DC needed)	532	1187
Proving Infeasible Plan	54	123

Table 5.1: Average and maximum runtime of the planner for various planning operations.

5.8 Conclusion

In this chapter we have laid the foundations of a deliberate acting system targeting robotic platforms. The cornerstone of the system is our expressive chronicle representation that provides rich temporal and hierarchical constructs to model the capabilities of an autonomous agent as well as the dynamics of the environment. As a result, chronicles are used as a central component for producing plans and monitoring their execution. The constraint-based representation of the planning system further provides an excellent basis for adapting the plan to react to unforeseen contingencies. The system was tested on a simulated robotic environment and demonstrated to efficiently perform online planning and acting.

A critical capability for the robustness of the system is to soundly reason on temporal uncertainty under various observability scenarios. An equally important function in order to act in a partially observable environment is the capacity of the system to consider conservative models of high level actions that will be refined, while acting, into more detailed primitives.

Chapter 6
Conclusion

In this thesis, we have provided a comprehensive view of a system for deliberate acting in a robotic context.

We have presented a rich model for encoding the capabilities of an autonomous agent that generalizes the generative and HTN planning models in a single formalism. The model further provides rich semantics for encoding the temporal aspect of a planning domain. We developed a planning procedure capable of handling this model. The resulting planner uses a constraint-based procedure to explore the space of lifted partial plans. While the planner already presents good performance in a fully lifted setting, we introduced further techniques for automated reasoning on a ground version of the problem. Those techniques improve the scalability of the planner, making it competitive with state of the art temporal planners in a domain independent setting. Beyond the capacity to encode domain-specific constraints and procedural knowledge, our support for task hierarchies allows to enhance the scalability of the planner by providing domain-dependent guidance.

Those core capabilities of the planner have been extended to provide robust plans in the presence of temporal uncertainty under various observability assumptions. In this setting, the system is able to reason online on the observations needed to maintain its plan controllable and adapt its behavior accordingly.

Last, we have discussed the integration of this planning framework in a more general acting architecture that closes the gap between the high level strategies provided by a planner and the capabilities of a robotic platform. This is done by leveraging our rich temporal model for dispatching and monitoring while relying on the planner's capacity of repairing an inconsistent plan to react to unforeseen contingencies.

Perspectives and Future Work. There are several directions for extending this work.

First, the planning system currently supports a rather focused model and targets an efficient handling of temporal, hierarchical and causal relationships. The overall system would benefit from an extended set of capabilities. Handling resources and numeric fluents is one such capability that is prominent in many real world problems that we have disregarded in this thesis. Most constraint-based planning systems support such capabilities and there is an extensive literature on how such support could be integrated.

Second, most of our work has been based on the assumption that it is safe to rely on deterministic models of the environment (other than temporal uncertainty) when deciding for the high-level strategy of the system. Indeed in our framework, uncertainty is either handled locally by skill handlers or by adapting the plan after detecting a discrepancy with our model. In many cases, this assumption does not hold and uncertainty needs to be accounted for in the overall strategy of the agent.

For both planning and acting, the system faces the problem of refining abstract actions into more concrete ones. From the planner's point of view, this means refining high-level actions into primitive ones while for acting it means refining the planner's primitive actions into lower level commands. In both cases, this raises the question of when the

system should refine an abstract action. Indeed, if the system is confident in its capacity to achieve a given task, it is unnecessary to have a detailed strategy for achieving it. For instance, if I have a flexible schedule, it might be unnecessary to plan the details of a trip to another city. However, if my schedule is tight, it might necessary to plan all details to make sure that I can make it in time to fulfill my other constraints. A similar capability was provided in our last chapter by manually giving a conservative model to an high-level action. Such a conservative model would only be applicable if the system had sufficient resources available, otherwise, it would need to come up with a finer grained plan to make sure that the system would indeed be able to carry out the task. Ideally, given a formal hierarchical description of the problem, the system should be able to decide which task should be refined and when to ensure that *(i)* the system always keeps a valid, potentially abstract, plan, *(ii)* no computational resource is wasted on committing early to unnecessary refinements, and *(iii)* all actions are refined in time to allow for a delay free execution. The current state of affairs in planning is mostly unsatisfactory with this regard as task planners tend to always come up with unnecessary detailed plan. On the other hand, most acting systems are only reactive and wait until the last second to refine actions into primitive commands.

Last, much work remains to be done for extending the general capabilities of the overall acting system. One such extension is the capacity of the system to reason on its goals by abandoning the irrelevant or impossible ones as well as generating new ones to react to the perceived state of the environment. Monitoring is another key capability whose support is currently mainly delegated to specific skill handlers. A more formal model would be needed in order for the system to decide when it is safe or desirable to interrupt an action given the overall state of the system. Considering multi-agent system with communication constraints would also require additional work both at the planning level to ensure that a plan is indeed controllable given the communication constraints and at the acting level to ensure sufficient exchange of information for maintaining a shared plan.

Taking a step back, the choice of of developing an entirely new planning and acting system based on a non-strandard model was indeed a challenging task to accomplish in three years. The many domains spanned by this tool, including planning, acting, robotics and constraint programming, indeed required a large amount of work, both to build a deep theoretical and practical understanding of the domains as well as for the implementation of the many aspects of the system. It further lead to the difficulty in communicating about such a non-incremental development and, as a result, a large part of the work remains unpublished at the time of this writing. Nevertheless, we are confident in saying that the unique combination of features of the system and its overall efficiency was made possible by taking into account their many requirements when designing of the system.

References

[AK83] J. Allen and J. A. Koomen. "Planning Using a Temporal World Model". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1983, pp. 741–747 (cit. on p. 22).

[Alf+12] R. Alford, V. Shivashankar, U. Kuter, and D. S. Nau. "HTN Problem Spaces: Structure, Algorithms, Termination". In: *International Symposium on Combinatorial Search (SoCS)*. 2012, pp. 2–9 (cit. on p. 16).

[Alf+16] R. Alford, V. Shivashankar, M. Roberts, J. Frank, and D. W. Aha. "Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press. New York City, NY, USA, July 2016 (cit. on pp. 19, 24, 72).

[All83] J. Allen. "Maintaining Knowledge About Temporal Intervals". In: *Communications of the ACM* 26.11 (1983), pp. 823–832 (cit. on p. 22).

[Bar+12] J. Barreiro, M. Boyce, M. B. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. H. Morris, J. Ong, E. Remolina, T. Smith, and D. E. Smith. "EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization". In: *International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*. 2012, p. 24 (cit. on pp. 22, 74, 122).

[BCC12] J. Benton, A. Coles, and A. Coles. "Temporal Planning with Preferences and Time-Dependent Continuous Costs". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2012 (cit. on pp. 68, 79, 81).

[BD02] G. Boella and R. Damiano. "A Replanning Algorithm for a Reactive Agent Architecture". In: *Artificial Intelligence: Methodology, Systems, and Applications* (2002), pp. 183–192 (cit. on p. 120).

[BD06] T. Bailey and H. Durrant-Whyte. "Simultaneous localization and mapping (SLAM): part II". In: *Robotics Automation Magazine* 13.3 (2006), pp. 108–117 (cit. on p. 120).

[BD13] J. C. Boerkoel and E. H. Durfee. "Distributed Reasoning for Multiagent Simple Temporal Problems". In: *Journal of Artificial Intelligence Research (JAIR)* 47 (2013), pp. 95–156 (cit. on p. 121).

[Bec+14] P. Bechon, M. Barbier, G. Infantes, C. Lesire, and V. Vidal. "HiPOP: Hierarchical Partial-Order Planning". In: *European Starting AI Researcher Symposium (STAIRS)*. 2014 (cit. on pp. 17, 23, 30, 72).

[Bec+15] P. Bechon, M. Barbier, C. Lesire, G. Infantes, and V. Vidal. "Using Hybrid Planning for Plan Reparation". In: *ECMR*. 2015 (cit. on p. 17).

[Bec16] P. Bechon. "Planification multirobot pour des missions de surveillance avec contraintes de communication". PhD thesis. Université de Toulouse, 2016 (cit. on pp. 23, 72, 73, 120).

[Ber+00] D. Bernard, G. A. Dorais, E. Gamble, B. Kanefsky, J. A. Kurien, W. Millar, N. Muscettola, P. P. Nayak, N. Rouquette, K. Rajan, B. Smith, W. Taylor, and Y.-W. Tung. *Remote Agent Experiment DS1: Technology Validation Report*. Tech. rep. 2000 (cit. on p. 119).

[Ber+16] P. Bercher, H. Daniel, G. Behnke, and S. Biundo. “More than a Name? On Implications of Preconditions and Effects of Compound HTN Planning Tasks”. In: *European Conference on Artificial Intelligence (ECAI)*. 2016 (cit. on pp. 16, 17).

[BF97] A. L. Blum and M. L. Furst. “Fast Planning through Planning Graph Analysis”. In: *Artificial Intelligence* 90.1–2 (1997), pp. 281–300 (cit. on p. 69).

[BFL14] J. Bajada, M. Fox, and D. Long. “Temporal Plan Quality Improvement and Repair using Local Search”. In: *European Starting AI Researcher Symposium (STAIRS)*. 2014, pp. 41–50 (cit. on p. 119).

[BG01] B. Bonet and H. Geffner. “Planning as Heuristic Search”. In: *Artificial Intelligence* 129.1-2 (2001), pp. 5–33 (cit. on p. 69).

[BG99] B. Bonet and H. Geffner. “Planning as Heuristic Search: New Results”. In: *European Conference on Planning (ECP)*. 1999 (cit. on p. 69).

[BGI16] A. Bit-Monnot, M. Ghallab, and F. Ingrand. “Which Contingent Events to Observe for the Dynamic Controllability of a Plan”. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. New York, NY, United States, July 2016 (cit. on p. 3).

[Bid+15] J. Bidot, L. Karlsson, F. Lagriffoul, and A. Saffiotti. “Geometric Backtracking for Combined Task and Motion Planning in Robotic Systems”. In: *Artificial Intelligence* 1 (2015), pp. 1–37 (cit. on p. 122).

[BK00] F. Bacchus and F. Kabanza. “Using Temporal Logics to Express Search Control Knowledge for Planning”. In: *Artificial Intelligence* 116.1-2 (2000), pp. 123–191 (cit. on p. 68).

[BKB14] P. Bercher, S. Keen, and S. Biundo. “Hybrid Planning Heuristics Based on Task Decomposition Graphs”. In: *International Symposium on Combinatorial Search (SoCS)*. 2014 (cit. on pp. 17, 72).

[BM94] M. Beetz and D. McDermott. “Improving Robot Plans During Their Execution”. In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 1994, pp. 7–12 (cit. on p. 121).

[BN95] C. Bäckström and B. Nebel. “Complexity Results for SAS+ Planning”. In: *Computational Intelligence* 11.4 (1995), pp. 625–655 (cit. on pp. 8, 23).

[Boh+11] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, and S. Holzer. “Towards Autonomous Robotic Butlers: Lessons Learned with the PR2”. In: *International Conference on Robotics and Automation (ICRA)*. 2011, pp. 5568–5575 (cit. on p. 122).

[Bon+97] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. Slack. “Experiences with an Architecture for Intelligent, Reactive Agents”. In: *Journal of Experimental and Theoretical Artificial Intelligence* 9.2/3 (Apr. 1997), pp. 237–256 (cit. on p. 121).

[BS07] S. Bernardini and D. E. Smith. “Developing Domain-Independent Search Control for EUROPA2”. In: *ICAPS Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*. 2007 (cit. on p. 75).

[BS08] S. Bernardini and D. E. Smith. "Automatically Generated Heuristic Guidance for EUROPA2". In: *ISAIRAS*. 2008 (cit. on p. 75).

[BS09] S. Bernardini and D. E. Smith. "Towards Search Control via Dependency Graphs in Europa2". In: (2009) (cit. on p. 75).

[BS11] S. Bernardini and D. E. Smith. "Automatic Synthesis of Temporal Invariants". In: *Symposium on Abstraction, Reformulation and Approximation (SARA)*. 2011, pp. 10–17 (cit. on p. 23).

[BSB08] J. Bidot, B. Schattenberg, and S. Biundo. "Plan Repair in Hybrid Planning". In: *Künstliche Intelligenz (KI)*. Ed. by A. R. Dengel, K. Berns, T. M. Breuel, F. Bomarius, and T. R. Roth-Berghofer. Vol. 5243. Lecture Notes in Computer Science. Berlin, Heidelberg, 2008, pp. 169–176 (cit. on p. 120).

[BSD16a] A. Bit-Monnot, D. E. Smith, and M. B. Do. "Delete-free Reachability Analysis for Temporal and Hierarchical Planning". In: *ICAPS Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*. 2016, pp. 93–102 (cit. on p. 3).

[BSD16b] A. Bit-Monnot, D. E. Smith, and M. B. Do. "Delete-free Reachability Analysis for Temporal and Hierarchical Planning". In: *European Conference on Artificial Intelligence (ECAI)*. 2016 (cit. on p. 3).

[CA09] J. Choi and E. Amir. "Combining Planning and Motion Planning". In: *International Conference on Robotics and Automation (ICRA)*. 2009 (cit. on p. 122).

[CAG09] S. Cambon, R. Alami, and F. Gravot. "A Hybrid Approach to Intricate Motion, Manipulation and Task Planning". In: *International Journal of Robotics Research* 28 (2009), pp. 104–126 (cit. on p. 122).

[CAL92] R. Chatila, R. Alami, and H. Laruelle. "Integrated Planning and Execution Control of Autonomous Robot Actions". In: *International Conference on Robotics and Automation (ICRA)*. 1992 (cit. on p. 122).

[Cas+06] L. A. Castillo, J. Fdez-Olivares, Ó. Garc´*(i)*a-Pérez, and F. Palao. "Efficiently Handling Temporal Knowledge in an HTN Planner". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2006, pp. 63–72 (cit. on pp. 16, 17, 71).

[Cas+16] G. Casanova, C. Pralet, C. Lesire, and T. Vidal. "Solving Dynamic Controllability Problem of Multi-Agent Plans with Uncertainty Using Mixed Integer Linear Programming". In: *European Conference on Artificial Intelligence (ECAI)*. 2016, pp. 930–938 (cit. on pp. 99, 121).

[Ces+09] A. Cesta, G. Cortellessa, S. Fratini, and A. Oddi. "Developing an End-to-End Planning Application from a Timeline Representation Framework." In: *Innovative Applications of Artificial Intelligence Conference (IAAI)*. 2009 (cit. on pp. 22, 75, 122).

[CFG00] L. A. Castillo, J. Fdez-Olivares, and A. González. "A Hybrid Hierarchical Operator-based Planning Approach for the Design of Control Programs". In: *Workshop Planen und Konfigurieren (PuK)*. 2000 (cit. on p. 17).

[CFP08] A. Cesta, S. Fratini, and F. Pecora. "Unifying Planning and Scheduling as Timelines in a Component-based Perspective". In: *Archives of Control Science* 18.2 (2008), pp. 231–271 (cit. on p. 75).

[Chi+00a] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. "Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling". In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 2000, pp. 300–307 (cit. on pp. 77, 119, 121).

[Chi+00b] S. Chien, G. Rabideau, R. Knight, R. Sherwood, B. Engelhardt, D. Mutz, T. Estlin, B. Smith, F. Fisher, T. Barrett, G. Stebbins, and D. Tran. "ASPEN: Automated Planning and Scheduling for Space Mission Operations". In: *International Conference on Space Operations (SpaceOps)*. 2000, pp. 1–10 (cit. on pp. 22, 77).

[Chi+12] S. Chien, M. Johnston, J. Frank, M. Guiiano, A. Kavelaars, C. Lenzen, and N. Policella. "A Generalized Timeline Representation, Services, andInterface for Automating Space Mission Operations". In: *International Conference on Space Operations (SpaceOps)*. 2012, pp. 1–17 (cit. on pp. 23, 75, 77).

[Chi+98] S. Chien, B. Smith, G. Rabideau, N. Muscettola, and K. Rajan. "Automated Planning and Scheduling for Goal-Based Autonomous Spacecraft". In: *IEEE Intelligent Systems* (1998) (cit. on p. 77).

[CHP14] C. Combi, L. Hunsberger, and R. Posenato. "An Algorithm for Checking the Dynamic Controllability of a Conditional Simple Temporal Network with Uncertainty - Revisited". In: *Communications in Computer and Information Science* 449 (2014), pp. 314–331 (cit. on p. 99).

[Cim+14a] A. Cimatti, L. Hunsberger, A. Micheli, R. Posenato, and M. Roveri. "Sound and Complete Algorithms for Checking the Dynamic Controllability of Temporal Networks with Uncertainty, Disjunction and Observation". In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2014, pp. 27–36 (cit. on p. 99).

[Cim+14b] A. Cimatti, L. Hunsberger, A. Micheli, and M. Roveri. "Using Timed Game Automata to Synthesize Execution Strategies for Simple Temporal Networks with Uncertainty". In: *AAAI Conference on Artificial Intelligence*. 2014, pp. 2242–2249 (cit. on pp. 99, 121).

[Cim+16] A. Cimatti, L. Hunsberger, A. Micheli, R. Posenato, and M. Roveri. "Dynamic Controllability via Timed Game Automata". In: *Acta Informatica* (2016) (cit. on pp. 99, 121).

[Cir+14] M. Cirillo, F. Pecora, H. Andreasson, T. Uras, and S. Koenig. "Integrated Motion Planning and Coordination for Industrial Vehicles". In: (2014) (cit. on p. 77).

[CK05] W. Cushing and S. Kambhampati. "Replanning : a New Perspective". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2005 (cit. on p. 119).

[CMR10] M. C. Cooper, F. Maris, and P. Régnier. "Compilation of a High-Level Temporal Planning Language into PDDL 2.1". In: *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. Vol. 2. 2010, pp. 181–188 (cit. on p. 20).

[CMR12a] A. Cimatti, A. Micheli, and M. Roveri. "Solving Temporal Problems using SMT: Strong Controllability". In: *International Conference on Principles and Practice of Constraint Programming*. 2012, pp. 248–264 (cit. on p. 99).

[CMR12b] A. Cimatti, A. Micheli, and M. Roveri. "Solving Temporal Problems using SMT: Weak Controllability". In: *AAAI Conference on Artificial Intelligence*. 2012 (cit. on p. 99).

[CMR13] M. C. Cooper, F. Maris, and P. Régnier. "Managing Temporal Cycles in Planning Problems Requiring Concurrency". In: *Computational Intelligence* 29.1 (2013), pp. 111–128 (cit. on pp. 16, 20–22, 53, 165).

[CMR14] A. Cimatti, A. Micheli, and M. Roveri. "Solving Strong Controllability of Temporal Problems with Uncertainty using SMT". In: *Constraints* (2014) (cit. on p. 99).

[CMR15] A. Cimatti, A. Micheli, and M. Roveri. "An SMT-based approach to weak controllability for disjunctive temporal problems with uncertainty". In: *Artificial Intelligence* 224 (2015), pp. 1–27 (cit. on p. 99).

[CO96a] A. Cesta and A. Oddi. "DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains". In: *New Directions in AI Planning* (1996) (cit. on p. 22).

[CO96b] A. Cesta and A. Oddi. "Gaining Efficiency and Flexibility in the Simple Temporal Problem." In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 1996, pp. 45–50 (cit. on p. 40).

[Col+08] A. Coles, M. Fox, D. Long, and A. Smith. "Planning with Problems Requiring Temporal Coordination". In: *AAAI Conference on Artificial Intelligence*. 2008 (cit. on pp. 68, 80, 81).

[Col+10] A. Coles, A. Coles, M. Fox, and D. Long. "Forward-Chaining Partial-Order Planning". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2010, pp. 42–49 (cit. on pp. 23, 24, 28, 49, 68, 79, 81).

[Col+12] A. Coles, A. Coles, M. Fox, and D. Long. "COLIN: Planning with Continuous Linear Numeric Change". In: *Journal of Artificial Intelligence Research (JAIR)* 44 (2012), pp. 1–96 (cit. on pp. 23, 68, 81).

[CPL15] G. Casanova, C. Pralet, and C. Lesire. "Managing Dynamic Multi-Agent Simple Temporal Network". In: *AAMAS* (2015), pp. 1171–1179 (cit. on p. 121).

[CS02] S. Coradeschi and A. Saffiotti. "Perceptual Anchoring: a Key Concept for Plan Execution in Embedded Systems". In: *Advances in Plan-Based Control of Robotic Agents*. Ed. by M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack. 2002, pp. 89–105 (cit. on p. 120).

[Cus+07a] W. Cushing, S. Kambhampati, Mausam, and D. S. Weld. "When is Temporal Planning Really Temporal?" In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2007, pp. 1852–1859 (cit. on pp. 16, 20, 21, 68, 79).

[Cus+07b] W. Cushing, S. Kambhampati, K. Talamadupula, D. S. Weld, and Mausam. "Evaluating Temporal Planning Domains". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2007, pp. 1–8 (cit. on pp. 81, 84).

[Cus12] W. A. Cushing. "When is Temporal Planning Really Temporal?" PhD thesis. Arizon State University, 2012 (cit. on pp. 23, 24).

[DFP10] A. Dovier, A. Formisiano, and E. Pontelli. "Multivalued Action Languages with Constraints in CLP(FD)". In: *Theory and Practice of Logic Programming* 10.02 (2010), p. 167 (cit. on p. 23).

[DGG93] C. Dousson, P. Gaborit, and M. Ghallab. "Situation Recognition: Representation and Algorithms". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1993, pp. 166–174 (cit. on p. 120).

[DK03] M. B. Do and S. Kambhampati. "Sapa: A Scalable Multi-objective Heuristic Metric Temporal Planner". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 155–194 (cit. on p. 68).

[DKH09] P. Doherty, J. Kvarnström, and F. Heintz. "A temporal Logic-based Planning and Execution Monitoring Framework for Unmanned Aircraft Systems". In: *AAMAS* 19.3 (Feb. 2009), pp. 332–377 (cit. on p. 119).

[DL07] C. Dousson and P. Le Maigat. "Chronicle Recognition Improvement Using Temporal Focusing and Hierarchization". In: *IJCAI* (2007), pp. 324–329 (cit. on p. 120).

[DMP91] R. Dechter, I. Meiri, and J. Pearl. "Temporal Constraint Networks". In: *Artificial Intelligence* 49.1-3 (May 1991), pp. 61–95 (cit. on p. 40).

[DPS13] M. Di Rocco, F. Pecora, and A. Saffiotti. "When Robots are Late: Configuration Planning for Multiple Robots with Dynamic Goals". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Nov. 2013, pp. 9515–5922 (cit. on pp. 76, 77).

[DTB13] F. Dvorák, D. Toropila, and R. Barták. "Towards AI Planning Efficiency: Finite-domain State Variable Reformulation". In: *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*. 2013 (cit. on p. 23).

[DU11] M. B. Do and S. Uckun. "Timeline-based Planning System for Manufacturing Applications". In: *Artificial Intelligence and Logistics* 1 (2011), p. 13 (cit. on p. 22).

[Dvo+14a] F. Dvorák, R. Barták, A. Bit-Monnot, F. Ingrand, and M. Ghallab. "Planning and Acting with Temporal and Hierarchical Decomposition Models". In: *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. Nov. 2014, pp. 115–121 (cit. on p. 3).

[Dvo+14b] F. Dvorák, A. Bit-Monnot, F. Ingrand, and M. Ghallab. "A Flexible ANML Actor and Planner in Robotics". In: *ICAPS Workshop on Planning and Robotics (PlanRob)*. Portsmouth, United States, June 2014 (cit. on pp. 3, 81).

[ECW97] T. Estlin, S. Chien, and X. Wang. "An Argument for a Hybrid HTN/Operator-Based Approach to Planning". In: *European Conference on Planning (ECP)*. 1997, pp. 182–194 (cit. on p. 17).

[Ede03] S. Edelkamp. "Taming Numbers and Duration in the Model Checking Integrated Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 195–238 (cit. on p. 68).

[EH04] S. Edelkamp and J. Hoffmann. “PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition”. In: *International Planning Competition (IPC-2004)*. 2004 (cit. on p. 20).

[EHN94] K. Erol, J. A. Hendler, and D. S. Nau. “UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning”. In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 1994, pp. 249–254 (cit. on pp. 11, 16, 43, 70).

[Elk+12] M. Elkawkagy, P. Bercher, B. Schattenberg, and S. Biundo. “Improving Hierarchical Planning Performance by the Use of Landmarks.” In: *AAAI Conference on Artificial Intelligence*. 2012 (cit. on p. 72).

[EMR12] P. Eyerich, R. Mattmüller, and G. Röger. “Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning”. In: *Springer Tracts in Advanced Robotics* 76.STAR (2012), pp. 49–64 (cit. on pp. 23, 68, 79).

[EWH10] R. T. Effinger, B. C. Williams, and A. G. Hofmann. “Dynamic Execution of Temporally and Spatially Flexible Reactive Programs.” In: *Bridging the Gap Between Task and Motion Planning*. 2010 (cit. on p. 121).

[Fde+06] J. Fdez-Olivares, L. A. Castillo, Ó. Garc´*(i)*a-Pérez, and F. Palao. “Bringing Users and Planning Technology Together. Experiences in SIADEX”. In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2006 (cit. on p. 71).

[FGT03] M. Fichtner, A. Großmann, and M. Thielscher. “Intelligent Execution Monitoring in Dynamic Environments”. In: *Fundamenta Informaticae* (2003), pp. 1–22 (cit. on p. 119).

[Fik71] R. E. Fikes. “Monitored Execution of Robot Plans Produced by STRIPS”. In: *IFIP Congress*. Ljubljana, Yugoslavia, Aug. 1971 (cit. on p. 121).

[Fir87] R. J. Firby. “An Investigation into Reactive Planning in Complex Domains”. In: *AAAI Conference on Artificial Intelligence*. 1987 (cit. on p. 121).

[FJ03] J. Frank and A. Jónsson. “Constraint-Based Attribute and Interval Planning”. In: *Constraints* 8.4 (2003), pp. 339–364 (cit. on pp. 22, 73, 74).

[FL03] M. Fox and D. Long. “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains.” In: *Journal of Artificial Intelligence Research (JAIR)* (2003) (cit. on p. 23).

[FL08] A. Ferrein and G. Lakemeyer. “Logic-based Robot Control in Highly Dynamic Domains”. In: *Robotics and Autonomous Systems* 56.11 (2008), pp. 980–991 (cit. on p. 122).

[FLH04] M. Fox, D. Long, and K. Halsey. “An Investigation into the Expressive Power of PDDL2.1”. In: *European Conference on Artificial Intelligence (ECAI)*. 2004 (cit. on pp. 20, 70).

[Flo62] R. W. Floyd. “Algorithm 97: Shortest Path”. In: *Communications of the ACM*. 1962 (cit. on p. 40).

[Fox+06] M. Fox, A. Gerevini, D. Long, and I. Serina. “Plan Stability: Replanning versus Plan Repair”. In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2006, pp. 212–221 (cit. on p. 119).

[Fra+11] S. Fratini, A. Cesta, R. De Benedictis, A. Orlandini, and R. Rasconi. "APSI-based Deliberation in Goal Oriented Autonomous Controllers". In: *ASTRA*. 2011 (cit. on p. 75).

[FSW05] G. Fraser, G. Steinbauer, and F. Wotawa. "Plan Execution in Dynamic Environments". In: *IEA/AIE*. Ed. by M. Ali and F. Esposito. Vol. 3533. Lecture Notes in Computer Science. Berlin, Heidelberg, 2005, pp. 208–217 (cit. on p. 119).

[Fuk+97] A. Fukunaga, G. Rabideau, S. Chien, and D. Yan. "ASPEN: A Framework for Automated Planning and Scheduling of Spacecraft Control and Operations". In: *ISAIRAS*. 1997 (cit. on pp. 22, 77).

[GA15] I. Georgievski and M. Aiello. "HTN Planning: Overview, Comparison, and Beyond". In: *Artificial Intelligence* 222 (2015), pp. 124–156 (cit. on p. 71).

[GA83] M. Ghallab and D. G. Allard. "A-epsilon - An Efficient Near Admissible Heuristic Search Algorithm". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by A. Bundy. 1983, pp. 789–791 (cit. on p. 65).

[Gar15] C. R. Garrett. "FFRob: An Efficient Heuristic for Task and Motion Planning". In: *Algorithmic Foundations of Robotics*. 2015 (cit. on p. 122).

[GB11] T. Geier and P. Bercher. "On the Decidability of HTN Planning with Task Insertion". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by T. Walsh. 2011, pp. 1955–1961 (cit. on pp. 16, 17).

[Gha96] M. Ghallab. "On Chronicles: Representation, On-line Recognition and Learning." In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1996, pp. 597–606 (cit. on p. 120).

[GL05] A. Gerevini and D. Long. *Plan Constraints and Preferences in PDDL3*. Tech. rep. Department of Electronics for Automation, University of Brescia, 2005, pp. 1–12 (cit. on p. 20).

[GL94] M. Ghallab and H. Laruelle. "Representation and Control in IxTeT, a Temporal Planner". In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 1994, pp. 61–67 (cit. on pp. 16, 22, 29, 30, 73).

[GLA11] I. Georgievski, A. Lazovik, and M. Aiello. *Phantomization in an HTN Planner*. Tech. rep. University of Groningen, 2011, p. 12 (cit. on p. 19).

[GNT04] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. 2004, p. 663 (cit. on pp. 6, 29, 43, 81).

[Gol06] R. P. Goldman. "Durative Planning in HTNs". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. Vol. 6. 2005. 2006, pp. 382–385 (cit. on p. 71).

[GS00] A. Gerevini and I. Serina. "Fast Plan Adaptation through Planning Graphs: Local and Systematic Search Techniques". In: *Proc. of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-00)*. 2000, pp. 112–121 (cit. on p. 119).

[GSS03] A. Gerevini, A. Saetti, and I. Serina. "Planning Through Stochastic Local Search and Temporal Action Graphs in LPG". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 239–290 (cit. on pp. 69, 119).

[GSS06] A. Gerevini, A. Saetti, and I. Serina. "An Approach to Temporal Planning and Sheduling in Domains with Predictable Exogenous Events". In: *Journal of Artificial Intelligence Research (JAIR)* 25 (2006), pp. 187–231 (cit. on pp. 68, 69).

[GSS10] A. Gerevini, A. Saetti, and I. Serina. "Temporal Planning with Problems Requiring Concurrency through Action Graphs and Local Search". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2010, pp. 226–229 (cit. on p. 69).

[Has06] P. Haslum. "Improving Heuristics through Relaxed Search - An Analysis of TP4 and HSP*a in the 2004 Planning Competition". In: *Journal of Artificial Intelligence Research (JAIR)* 25 (2006), pp. 233–267 (cit. on pp. 68, 69).

[Haw11] N. Hawes. "A Survey of Motivation Frameworks for Intelligent Systems". In: *Artificial Intelligence* 175.5-6 (2011), pp. 1020–1036 (cit. on p. 122).

[HBL98] D. Häihnel, W. Burgard, and G. Lakemeyer. "GOLEX – Bridging the Gap between Logic (GOLOG) and a Real Robot". In: *Lecture Notes in Computer Science* (1998) (cit. on p. 122).

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1973 (cit. on p. 122).

[HCZ10] R. Huang, Y. Chen, and W. Zhang. "A Novel Transition Based Encoding Scheme for Planning as Satisfiability". In: *Artificial Intelligence* (2010), pp. 89–94 (cit. on p. 69).

[HDR08] M. Helmert, M. B. Do, and I. Refanidis. *Proceedings of the 2008 International Planning Competition, Deterministic Track*. Tech. rep. 2008 (cit. on p. 68).

[Hel06] M. Helmert. "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 26 (2006), pp. 191–246. arXiv: `arXiv:1109.6051v1` (cit. on pp. 23, 28, 69).

[Hel09] M. Helmert. "Concise Finite-domain Representations for PDDL Planning Tasks". In: *Artificial Intelligence* 173.5-6 (2009), pp. 503–535 (cit. on p. 23).

[HG00] P. Haslum and H. Geffner. "Admissable Heuristics for Optimal Planning". In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 2000, pp. 140–149 (cit. on p. 69).

[HG01] P. Haslum and H. Geffner. "Heuristic Planning with Time and Resources". In: *European Conference on Planning (ECP)*. 2001, pp. 107–112 (cit. on pp. 20, 68, 69).

[HG08] M. Helmert and H. Geffner. "Unifying the Causal Graph and Additive Heuristics". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2008 (cit. on pp. 69, 79).

[HKD10] F. Heintz, J. Kvarnström, and P. Doherty. "Bridging the Sense-Reasoning Gap: DyKnow – Stream-based Middleware for Knowledge Processing". In: *Advanced Engineering Informatics* 24.1 (Jan. 2010), pp. 14–26 (cit. on p. 120).

[HLF04] K. Halsey, D. Long, and M. Fox. “CRIKEY - A Temporal Planner Looking at the Integration of Scheduling and Planning”. In: *Workshop on Integrating Planning into Scheduling* 02 (2004), p. 46 (cit. on p. 68).

[HN01] J. Hoffmann and B. Nebel. “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), pp. 253–302 (cit. on p. 69).

[Hof01] J. Hoffmann. “FF: The Fast-Forward Planning System”. In: *AI Magazine* 22.3 (2001), pp. 57–62 (cit. on p. 49).

[Hof03] J. Hoffmann. “The Metric-FF Planning System: Translating ”Ignoring Delete Lists” to Numeric State Variables”. In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 291–341. arXiv: 1106.5271 (cit. on p. 68).

[Hun09] L. Hunsberger. “Fixing the Semantics for Dynamic Controllability and Providing a more Practical Characterization of Dynamic Execution Strategies”. In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2009, pp. 155–162 (cit. on p. 98).

[Hun10] L. Hunsberger. “A Fast Incremental Algorithm for Managing the Execution of Dynamically Controllable Temporal Networks”. In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2010, pp. 121–128 (cit. on pp. 98, 121).

[Hun13] L. Hunsberger. “A Faster Execution Algorithm for Dynamically Controllable STNUs”. In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2013, pp. 26–33 (cit. on p. 98).

[Hun14] L. Hunsberger. “New Techniques for Checking Dynamic Controllability of Simple Temporal Networks with Uncertainty”. In: *International Conference on Agents and Artificial Intelligence (ICAART)*. Vol. 8946. 2014, pp. 63–73 (cit. on p. 98).

[Hun16] L. Hunsberger. “Efficient Execution of Dynamically Controllable Simple Temporal Networks with Uncertainty”. In: *Acta Informatica* 53.2 (2016), pp. 89–147 (cit. on pp. 98, 121).

[IG14] F. Ingrand and M. Ghallab. “Deliberation for Autonomous Robots: A Survey”. In: *Artificial Intelligence* 1 (2014), pp. 1–35 (cit. on p. 118).

[Ing+96] F. Ingrand, R. Chatila, R. Alami, and F. Robert. “PRS : A High Level Supervision and1 Control Language for Autonomous Mobile Robots”. In: *International Conference on Robotics and Automation (ICRA)*. 1996, pp. 43–49 (cit. on p. 121).

[IRW99] M. Ingham, R. Ragno, and B. C. Williams. “A Reactive Model-based Programming Language for Robotic Space Explorers Model-based Programming”. In: *ISAIRAS*. 1999 (cit. on p. 121).

[JB98] P. Jonsson and C. Bäckström. “State-Variable Planning under Structural Restrictions: Algorithms and Complexity”. In: *Artificial Intelligence* 100.1-2 (1998), pp. 125–176 (cit. on p. 23).

[Kam93] S. Kambhampati. "On the Utility of Systematicity: Understanding Tradeoffs Between Redundancy and Commitment in Partial-Order Planning". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1993, pp. 110–112 (cit. on pp. 45, 46).

[Kar+08] L. Karlsson, A. Bouguerra, M. Broxvall, S. Coradeschi, and A. Saffiotti. "To Secure an Anchor – A Recovery Planning Approach to Ambiguity in Perceptual Anchoring". In: *AI Communications* 21.1 (2008), pp. 1–14 (cit. on p. 120).

[KKY95] S. Kambhampati, C. A. Knoblock, and Q. Yang. "Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial Order Planning". In: *Artificial Intelligence* 76.602 (1995), pp. 167–238 (cit. on pp. 30, 44).

[KL11] L. P. Kaelbling and T. Lozano-Perez. "Hierarchical Task and Motion Planning in the Now". In: *International Conference on Robotics and Automation (ICRA)*. 2011, pp. 1470–1477 (cit. on p. 122).

[KMS98] S. Kambhampati, A. Mali, and B. Srivastava. "Hybrid Planning for Partially Hierarchical Domains". In: *AAAI Conference on Artificial Intelligence*. c. 1998, pp. 882–888 (cit. on pp. 17, 18).

[Kno94] C. A. Knoblock. "Automatically Generating Abstractions for Planning". In: *Artificial Intelligence* 68.2 (Aug. 1994), pp. 243–302 (cit. on p. 66).

[kR96] A. El-kholy and B. Richards. "Temporal and Resource Reasoning in Planning: the parcPLAN approach". In: *European Conference on Artificial Intelligence (ECAI)*. 1996, pp. 614–618 (cit. on p. 22).

[Krü+07] V. Krüger, D. Kragic, A. Ude, and C. Geib. "The Meaning of Action: A Review on Action Recognition and Mapping". In: *Advanced Robotics* 21.13 (2007), pp. 1473–1501 (cit. on p. 120).

[KSH06] H. Kautz, B. Selman, and J. Hoffmann. "SATPlan: Planning as Satisfiability". In: *Booklet of the 2006 International Planning Competition* (2006) (cit. on p. 69).

[KW05] R. V. D. Krogt and M. D. Weerdt. "Plan Repair as an Extension of Planning". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2005, pp. 161–170 (cit. on p. 119).

[KWA01] P. Kim, B. C. Williams, and M. Abramson. "Executing Reactive, Model-based Programs through Graph-based Temporal Planning." In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2001, pp. 487–493 (cit. on p. 121).

[Lan+15] A. Lanz, R. Posenato, C. Combi, and M. Reichert. "Simple Temporal Networks with Partially Shrinkable Uncertainty". In: *International Conference on Agents and Artificial Intelligence (ICAART)*. 2015 (cit. on p. 99).

[Lem+10] S. Lemaignan, R. Ros, M. Lorenz, R. Alami, and M. Beetz. "ORO, a Knowledge Management Platform for Cognitive Architectures in Robotics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2010, pp. 3548–3553 (cit. on pp. 124, 137).

[Lem04] S. Lemai-Chenevier. "IXTET-EXEC: Planning, Plan Repair and Execution Control with Time and Resource Management". PhD thesis. 2004 (cit. on p. 120).

[LF03] D. Long and M. Fox. "Exploiting a Graphplan Framework in Temporal Planning". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. Vol. 1. 2003, pp. 52–61 (cit. on p. 69).

[LI04] S. Lemai and F. Ingrand. "Interleaving Temporal Planning and Execution in Robotics Domains". In: *AAAI Conference on Artificial Intelligence*. 2004, pp. 617–622 (cit. on p. 120).

[LW14] S. J. Levine and B. C. Williams. "Concurrent Plan Recognition and Execution for Human-Robot Teams". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2014 (cit. on pp. 121, 122).

[Mac77] A. K. Mackworth. "Consistency in Networks of Relations". In: *Artificial Intelligence* (1977) (cit. on p. 41).

[McD+98] D. McDermott, M. Ghallab, A. E. Howe, C. A. Knoblock, A. Ram, M. M. Veloso, D. Weld, and D. E. Wilkins. *PDDL: the Planning Domain Definition Language*. Tech. rep. Yale Center for Computational Vision and Control, 1998 (cit. on p. 20).

[McG+08] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. "A deliberative architecture for AUV control". In: *International Conference on Robotics and Automation (ICRA)*. May 2008, pp. 1049–1054 (cit. on pp. 77, 122).

[McG+09] C. McGann, E. Berger, J. Boren, S. Chitta, B. Gerkey, S. Glaser, E. Marder-Eppstein, B. Marthi, W. Meeussen, T. Pratkanis, E. Marder-Eppstein, and M. Wise. "Model-based, Hierarchical Control of a Mobile Manipulation Platform". In: *ICAPS Workshop on Planning and Plan Execution for Real World Systems*. 2009 (cit. on p. 122).

[MM05] P. H. Morris and N. Muscettola. "Temporal Dynamic Controllability Revisited". In: *AAAI Conference on Artificial Intelligence*. 2005, pp. 1193–1198 (cit. on p. 98).

[MMT98] N. Muscettola, P. H. Morris, and I. Tsamardinos. "Reformulating Temporal Plans for Efficient Execution." In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1998, pp. 444–452 (cit. on p. 120).

[MMV01] P. H. Morris, N. Muscettola, and T. Vidal. "Dynamic Control of Plans with Temporal Uncertainty". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2001, pp. 494–502 (cit. on pp. 98, 102, 109).

[MO15] M. C. Mayer and A. Orlandini. "An Executable Semantics of Flexible Plans in Terms of Timed Game Automata". In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2015, pp. 160–169 (cit. on pp. 99, 121).

[Mof07] M. D. Moffitt. "On the Partial Observability of Temporal Uncertainty". In: *AAAI Conference on Artificial Intelligence*. 2007, pp. 1031–1037 (cit. on p. 99).

[Mon+03] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. "FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2003, pp. 1151–1156 (cit. on p. 120).

[Mor14] P. H. Morris. "Dynamic Controllability and Dispatchability Relationships". In: *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*. 2014 (cit. on pp. 98, 104, 108, 121, 130).

[MP14] M. Mansouri and F. Pecora. "More Knowledge on the Table : Planning with Space , Time and Resources for Robots". In: *International Conference on Robotics and Automation (ICRA)*. 2014 (cit. on pp. 76, 77).

[MPM94] R. M. Michael, M. E. Pollack, and J. D. Moore. "Decomposition in Partial-Order Planning". In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 1994 (cit. on p. 17).

[MR08] F. Maris and P. Régnier. "TLP-GP: Solving Temporally-expressive Planning Problems". In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2008 (cit. on p. 69).

[MR91] D. McAllester and D. Rosenblitt. "Systematic Nonlinear Planning". In: *AAAI Conference on Artificial Intelligence*. 1991 (cit. on pp. 45, 46, 70).

[MSR16] J. K. Mogali, S. F. Smith, and Z. B. Rubinstein. "Distributed Decoupling Of Multiagent Simple Temporal Problems". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2016, pp. 408–415 (cit. on p. 121).

[Mus+00] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. "A Unified Approach to Model-Based Planning and Execution". In: *IAS*. 2000 (cit. on p. 122).

[Mus+02] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. "IDEA: Planning at the Core of Autonomous Reactive Agents". In: *International NASA Workshop on Planning and Scheduling for Space*. Vol. 1. May. 2002, p. 7 (cit. on pp. 22, 122).

[Mus+98] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. "Remote Agent: to Boldly Go Where no AI System Has Gone Before". In: *Artificial Intelligence* 103.1-2 (1998), pp. 5–47 (cit. on p. 119).

[Mus94] N. Muscettola. "HSTS: Integrating Planning and Scheduling". In: *Intelligent Scheduling* (1994), pp. 169–212 (cit. on pp. 73, 74).

[Mye99] K. L. Myers. "CPEF: A Continuous Planning and Execution Framework". In: *AI Magazine* 20.4 (1999), pp. 63–70 (cit. on p. 121).

[Nau+00] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. *SHOP and M-SHOP: Planning with Ordered Task Decomposition*. Tech. rep. Computer Science Departement, University of Maryland, 2000 (cit. on p. 71).

[Nau+03] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. "SHOP2: An HTN Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 379–404 (cit. on pp. 11, 16–18, 29, 46, 71).

[Nay+99] P. P. Nayak, J. Kurien, G. A. Dorais, W. Millar, K. Rajan, R. Kanefsky, E. D. Bernard, B. E. Gamble Jr, N. Muscettola, N. Rouquette, and D. B. Smith. "Validating the DS-1 Remote Agent Experiment". In: *ISAIRAS*. Vol. 440. 1999, p. 349 (cit. on p. 74).

[NKD13] M. Nilsson, J. Kvarnström, and P. Doherty. "Incremental Dynamic Controllability Revisited". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2013, pp. 337–341 (cit. on pp. 98, 121).

[NKD14a] M. Nilsson, J. Kvarnström, and P. Doherty. "Classical Dynamic Controllability Revisited". In: *International Conference on Agents and Artificial Intelligence (ICAART)*. 2014, pp. 130–141 (cit. on pp. 121, 129).

[NKD14b] M. Nilsson, J. Kvarnström, and P. Doherty. "Incremental Dynamic Controllability in Cubic Worst-Case Time". In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2014, pp. 17–26 (cit. on p. 98).

[NKD15] M. Nilsson, J. Kvarnström, and P. Doherty. "Efficient Processing of Simple Temporal Networks with Uncertainty: Algorithms for Dynamic Controllability Verification". In: *Acta Informatica* (Oct. 2015) (cit. on pp. 98, 121).

[Pec+12] F. Pecora, M. Cirillo, F. Dell'Osa, J. Ullberg, and A. Saffiotti. "A Constraint-based Approach for Proactive, Context-aware Human Support". In: *JAISE* 4.4 (2012), pp. 347–367 (cit. on p. 77).

[Pen93] J. S. Penberthy. "Planning with Continuous Change". PhD thesis. University of Washington, 1993 (cit. on p. 73).

[Pet05] O. Pettersson. "Execution Monitoring in Robotics: A survey". In: *Robotics and Autonomous Systems* 53.2 (2005), pp. 73–88 (cit. on p. 119).

[Pla08] L. Planken. *New Algorithms for the Simple Temporal Problem*. Tech. rep. Delft University, 2008 (cit. on p. 40).

[PRM10] F. Py, K. Rajan, and C. McGann. "A Systematic Agent Framework for Situated Autonomous Systems". In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 2010 (cit. on p. 122).

[PW92] J. S. Penberthy and D. S. Weld. "UCPOP: A Sound, Complete, Partial Order Planner for ADL". In: *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 1992, pp. 103–114 (cit. on pp. 30, 45).

[PW94] J. S. Penberthy and D. S. Weld. "Temporal Planning with Continuous Change". In: *AAAI Conference on Artificial Intelligence*. Vol. 2. 1994, pp. 1010–1015 (cit. on p. 73).

[PWK08] L. Planken, M. de Weerdt, and R. van der Krogt. "P3C: A New Algorithm for the Simple Temporal Problem". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2008, pp. 256–263 (cit. on p. 40).

[PWY10] L. Planken, M. de Weerdt, and N. Yorke-Smith. "Incrementally Solving STNs by Enforcing Partial Path Consistency". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2010, pp. 129–136 (cit. on p. 40).

[Rab+99] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. "Iterative Repair Planning for Spacecraft operations using the ASPEN system". In: *ISAIRAS*. Vol. 440. 1999, p. 99 (cit. on p. 112).

[RF14] B. Ridder and M. Fox. "Heuristic Evaluation Based on Lifted Relaxed Planning Graphs". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2014 (cit. on p. 70).

[RG15] M. F. Rankooh and G. Ghassem-Sani. "ITSAT: An Efficient SAT-Based Temporal Planner". In: *Journal of Artificial Intelligence Research (JAIR)* 53 (2015), pp. 541–632 (cit. on pp. 23, 69).

[RW10] S. Richter and M. Westphal. "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks". In: *Journal of Artificial Intelligence Research (JAIR)* 39 (2010), pp. 127–177 (cit. on p. 72).

[SA98] R. G. Simmons and D. Apfelbaum. "A Task Description Language for Robot Control". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 1998 (cit. on p. 121).

[Sac75] E. D. Sacerdoti. "The Nonlinear Nature of Plans". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1975. arXiv: `arXiv:1011.1669v3` (cit. on pp. 16, 70).

[Sch+13] A. Schwerdfeger, M. Boddy, D. E. Smith, J. Frank, and C. Mcgann. *The ANML Language, Revised [DRAFT]*. Tech. rep. Adventium Labs, 2013 (cit. on p. 25).

[Sch09] B. Schattenberg. "Hybrid Planning & Scheduling". PhD thesis. Ulm University, 2009 (cit. on pp. 17, 30, 44, 72).

[SFC08] D. E. Smith, J. Frank, and W. Cushing. "The ANML Language". In: *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*. 2008 (cit. on pp. 6, 20, 24).

[Sha+07] J. A. Shah, J. Stedl, B. C. Williams, and P. Robertson. "A Fast Incremental Algorithm for Maintaining Dispatchability of Partially Controllable Plans". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2007, pp. 296–303 (cit. on p. 98).

[Shi+12] V. Shivashankar, U. Kuter, D. S. Nau, and R. Alford. "A hierarchical goal-based formalism and algorithm for single-agent planning". In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. June 2012, pp. 981–988 (cit. on p. 72).

[Shi+13] V. Shivashankar, R. Alford, U. Kuter, and D. S. Nau. "The GoDeL Planning System: A More Perfect Union of Domain-Independent and Hierarchical Planning". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2013 (cit. on p. 72).

[Shi+16] V. Shivashankar, R. Alford, M. Roberts, and D. W. Aha. "Cost-Optimal Algorithms for Planning with Procedural Control Knowledge". In: *European Conference on Artificial Intelligence (ECAI)*. 2016. arXiv: `1607.01729` (cit. on p. 72).

[Sil+14] L. de Silva, M. Gharbi, A. K. Pandey, and R. Alami. "A New Approach to Combined Symbolic-Geometric Backtracking in the Context of Human-Robot Interaction". In: *International Conference on Robotics and Automation (ICRA)*. 2014 (cit. on p. 122).

[Sim92] R. G. Simmons. "Concurrent Planning and Execution for Autonomous Robots". In: *IEEE Control Systems* (1992) (cit. on p. 121).

[Smi+98] B. Smith, R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga. "Representing Spacecraft Mission Planning Knowledge in ASPEN". In: *AIPS Workshop on Knowledge Acquisition* (1998), p. 15 (cit. on p. 77).

[Smi03] D. E. Smith. "The Case for Durative Actions: A Commentary on PDDL2.1." In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 149–154 (cit. on p. 20).

[Smi04] D. E. Smith. "Choosing Objectives in Over-Subscription Planning." In: *ICAPS* (2004), pp. 393–401 (cit. on p. 122).

[Sri+14] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel. "Combined Task and Motion Planning Through an Extensible Planner-Independent Interface Layer". In: *International Conference on Robotics and Automation (ICRA)*. L. 2014 (cit. on p. 122).

[Sto+15] S. Stock, M. Mansouri, F. Pecora, and J. Hertzberg. "Online Task Merging with a Hierarchical Hybrid Task Planner for Mobile Service Robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 6459–6464 (cit. on pp. 19, 76, 77, 84).

[SW14] P. H. R. Q. A. Santana and B. C. Williams. "Chance-Constrained Consistency for Probabilistic Temporal Plan Networks". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. 2014 (cit. on p. 122).

[SW99] D. E. Smith and D. S. Weld. "Temporal Planning with Mutual Exclusion Reasoning." In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1999, pp. 326–337 (cit. on pp. 20, 69).

[Tat76] A. Tate. *Project Planning Using a Hierarchic Non-Linear Planner*. Tech. rep. 25. Department of Artificial Intelligence, University of Edinburgh, 1976 (cit. on p. 46).

[Tat77] A. Tate. "Generating Project Network". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1977 (cit. on p. 16).

[TDK94] A. Tate, B. Drabble, and R. Kirby. "O-Plan2: An Architecture for Command, Planning and Control". In: *Intelligent Scheduling* (1994), pp. 1–26 (cit. on pp. 16, 17, 22, 29, 70).

[TMM98] I. Tsamardinos, N. Muscettola, and P. H. Morris. "Fast Transformation of Temporal Plans for Efficient Execution". In: *AAAI Conference on Artificial Intelligence*. 1998 (cit. on pp. 112, 120).

[TPW11] O. ten Thije, L. Planken, and M. D. Weerdt. "Maintaining Partial Path Consistency in STNs under Event-Incremental Updates". 2011 (cit. on p. 40).

[Vat+13] S. Vattam, M. Klenk, M. Molineaux, and D. W. Aha. "Breadth of Approaches to Goal Reasoning : A Research Survey". In: *Conference on Advances in Cognitive Systems: Workshop on Goal Reasoning*. 2013, p. 111 (cit. on p. 122).

[Ver+05] V. Verma, T. Estlin, A. K. Jonsson, C. Pasareanu, R. G. Simmons, and K. Tso. "Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences". In: *ISAIRAS*. 2005 (cit. on p. 122).

[Ver83] S. A. Vere. "Planning in Time: Windows and Durations for Activities and Goals." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 5.3 (Mar. 1983), pp. 246–67 (cit. on p. 73).

[VF99] T. Vidal and H. Fargier. "Handling Contingency in Temporal Constraint Networks: From Consistency to Controllabilities". In: *Journal of Experimental and Theoretical Artificial Intelligence* 11 (1999) (cit. on pp. 95, 96, 98, 101).

[VG06] V. Vidal and H. Geffner. "Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming". In: *Artificial Intelligence* 170.3 (2006), pp. 298–335 (cit. on pp. 30, 69).

[VG96] T. Vidal and M. Ghallab. "Dealing with Uncertain Durations In Temporal Constraint Networks Dedicated to Planning". In: *European Conference on Artificial Intelligence (ECAI)*. 1996, pp. 48–54 (cit. on p. 95).

[Vid00] T. Vidal. "A Unified Dynamic Approach for Dealing with Temporal Uncertainty and Conditional Planning." In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 2000, pp. 395–402 (cit. on p. 121).

[Vid04] V. Vidal. "A Lookahead Strategy for Heuristic Search Planning". In: *International Conference on Automated Planning and Shceduling (ICAPS)*. Vidal. 2004, pp. 150–159 (cit. on p. 68).

[Vid11a] V. Vidal. "CPT4: An Optimal Temporal Planner Lost in a Planning Competition without Optimal Temporal Track". In: *The Seventh International Planning Competition: Description of Participant Planners of the Deterministic Track* (2011), pp. 25–28 (cit. on p. 23).

[Vid11b] V. Vidal. "YAHSP2: Keep It Simple, Stupid". In: *International Planning Competition (IPC-2011)*. 2011 (cit. on p. 68).

[Vid14] V. Vidal. "YAHSP3 and YAHSP3-MT in the 8th International Planning Competition". In: *International Planning Competition (IPC-2014)*. 2014 (cit. on p. 68).

[VR98] M. M. Veloso and P. Rizzo. "Mapping Planning Actions and Partially-Ordered into Execution Knowledge Plans". In: *Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*. 1998 (cit. on p. 121).

[War+07] I. Warfield, C. Hogg, S. Lee-Urban, and H. Muñoz-Avila. "Adaptation of Hierarchical Task Network Plans". In: *International Florida Artificial Intelligence Research Society Conference* (2007), pp. 429–434 (cit. on p. 120).

[War62] S. Warshall. “A Theorem on Boolean Matrices”. In: *Journal of the ACM* (1962) (cit. on p. 40).

[WGA16] J. Waldhart, M. Gharbi, and R. Alami. “A Novel Software Combining Task and Motion Planning for Human-Robot Interaction”. In: *AAAI Fall Symposia*. 2016 (cit. on p. 139).

[Wil90] D. E. Wilkins. “Can AI Planner Solve Practical Problems?” In: *Computational Intelligence* 6.4 (1990), pp. 232–246 (cit. on pp. 17, 70, 121).

[WM95] D. E. Wilkins and K. Myers. “A Common Knowledge Representation for Plan Generation and Reactive Execution”. In: *Journal of Logic and Computation* (1995) (cit. on pp. 16, 17, 70, 121).

[WSH16] M. Wehrle, S. Sievers, and M. Helmert. “Graph-Based Factorization of Classical Planning Problems”. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2016 (cit. on p. 70).

[XC03] L. Xu and B. Y. Choueiry. “A New Efficient Algorithm for Solving the Simple Temporal Problem.” In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2003, pp. 212– (cit. on p. 40).

[YN00] F. Yaman and D. S. Nau. “TimeLine: An HTN Planner that Can Reason about Time”. In: *AIPS Workshop on Planning in Temporal Domains*. 2000, p. 75 (cit. on p. 71).

[YS03] H. L. S. Younes and R. G. Simmons. “VHPOP: Versatile Heuristic Partial Order Planner”. In: *Journal of Artificial Intelligence Research (JAIR)* (2003) (cit. on pp. 30, 69, 120).

[Zwe+93] M. Zweben, E. Davis, B. Daun, and M. J. Deale. “Scheduling and Rescheduling with Iterative Repair”. In: *IEEE Transactions on Systems, Man and Cybernetics* 23.6 (1993), pp. 1588–1596 (cit. on p. 77).

Appendices

Appendix A

Overlength Proofs

A.1 Proof of Proposition 3.3.1

We first suppose that all actions have a single condition. With this assumptions, there are two sets of trivially reachable elements: fluents in timed initial literals and actions and fluents appearing in self-supporting causal loops. An action/fluent n is reachable if there is a path (i.e. sequence of actions/fluents) from one of those trivially reachable elements to n. Let us note that all fluents in timed initial literals are part of the assumed reachable set. Furthermore, a self-supporting causal loop necessarily contains an action with an after-condition [CMR13]. Since after-conditions are ignored at first, it means that this action will part of the assumed reachable set. Furthermore, propagation will never remove those from the assumed reachable set because we have an upper bound on their earliest appearance. We call this the set of truly reachable elements $E^{TrulyReach}$ which is a subset of the set of assumed reachable elements $E^{rAssReach}$. We define as $E^{NotReach}$ the subset of $E^{rAssReach}$ that is not reachable.

An element d is reachable if an element $s \in E^{TrulyReach}$, such that there is a path from s to d. The true earliest appearance of d is given by $\min_{s \in E^{TrulyReach}} ea(s) + spp(s, d)$, where $spp(s, d)$ is the length of the shortest path from s to d.

At first, our algorithm is optimistic which means that we consider as reachable all nodes with a path from an element of $E^{TrulyReach} \cup E^{NotReach}$. To show that earliest appearances of reachable nodes eventually converge to their true values, we first show that the earliest appearances of nodes in $E^{NotReach}$ indefinitely increase until their are removed from the model.

A node $n \in E^{NotReach}$ is necessarily an action with an after-condition that was optimistically ignored. The fact that n is not reachable means that its after-condition p is not reachable, meaning that there is not path form a an element of $E^{TrulyReach}$ to p. If p is not assumed reachable, then n will be removed. Otherwise we can distinguish two cases depending on which node in $E^{NotReach}$ provides the earliest start time of p:

- $ea(p) = ea(n) + spp(n, p)$. In this case, this is an unfeasible causal loop involving n and the earliest appearance of n will be increased.

- there is another node $n' \in E^{NotReach}$ such that $ea(p) = ea(n') + spp(n', p)$. In this case, we can recursively do a similar reasoning on n': it is either part of an unfeasible causal loop or depend an a node $n'' \in E^{NotReach}$. In both cases, it depends on a node involved in a causal loop and its earliest appearance would be increased meaning that the earliest appearances of p and then n would increase as well.

We have shown that the earliest appearances of all nodes in $E^{NotReach}$ indefinitely increase until they are removed. This is also the case of all unreachable nodes that were once assumed reachable because the sources of all their shortest path is a node in $E^{NotReach}$.

On the other hand, we will eventually reach a point where:

$$\min_{s \in E^{TrulyReach}} ea(s) + spp(s,d) = \min_{s' \in E^{TrulyReach} \cup E^{NotReach}} ea(s') + spp(s',d)$$

because the the earliest appearance of any $s' \in E^{NotReach}$ diverges towards infinity. As a consequence, earliest appearances of reachable nodes will eventually converge to their true values.

This result can be extended to actions with more than on condition by observing that, in a given iteration, the algorithm only uses a single condition of the action: the one that would make it the latest.

A.2 Proof of Proposition 3.3.2

The proof is split in several definitions and proposition to facilitate the reading. We use a graph formalism for the rest of this proof: a node is either a fluent or an an elementary action. We say that there is an edge from x to y if y is x is an action with effect y or if y is an action with condition x. Each edge e has a label $lbl(e)$ that is the delay from the condition to the action or from the action to the effect.

A.2.1 Self-dependent set

We first identify sufficient conditions to declare a set of nodes non-reachable. A node X is a predecessor of a node Y (noted $pred(Y) = X$) if the latest value of $ea(Y)$ was updated by an edge from X to Y. This is similar to the predecessor labels propagated in a Dijkstra algorithm. While Algorithm 2 does not maintain this information explicitly, it would be easy to add a predecessor field for each that would be updated every time the earliest appearance is modified.

Definition A.2.1 (Predecessor cycle)**.** A *predecessor cycle* is sequence $A_1 \to a_1 \to A_2 \to a_2 \dots A_n \to a_n \to A_1$ of edges where the source of an edge is the predecessor of its target (e.g. $pred(a_1) = A_1$). Upper case nodes are action nodes and lower case nodes are fluents.

Proposition A.2.1. *A predecessor cycle is of strictly positive length (i.e. the sum of the labels on the edges is strictly greater than 0).*

Proof. A cycle of predecessor means that an update of the first element (e.g. A_1) triggered an update of its direct successors (e.g. a_1) and all its indirect successors (e.g. A_2, a_2, a_n) including itself. Since the earliest appearance can only be increased as a result of an update, then the cycle has a strictly positive length (otherwise it would not have resulted in a greater value). □

Proposition A.2.2. *In a predecessor cycle, at least one effect edge $A_i \to a_i$ can be removed without altering the problem.*

Proof. A predecessor cycle represents an invalid combination of first achievers of fluents in the cycle. It means that having the action A_i as the first achiever for the fluent a_i (for all $i \in [1, n]$) would result in the condition that starting A_1 at a given time requires that A_1 had started at an earlier time. This is trivially non-possible hence at least one fluent a_k in the cycle must be first achieved by an action other than A_k. Since we are dealing with a delete-free model, the effect a_k can be removed from A_k without altering the problem. □

Definition A.2.2 (Self-dependent set)**.** We say that a set Ω of action and fluent nodes is a *self-dependent set* if:

- All nodes in Ω have a predecessor in Ω.
- For any fluent f in Ω, all achievers of f are in Ω.

It should be noted that the first element of Definition A.2.2 implies that *(i)* all elements of a self-dependent set have been updated at least once *(ii)* all actions in Ω depend on at least one fluent in Ω.

Proposition A.2.3. *If a node n is part of a self-dependent set Ω, then n is not possible.*

Proof. We first show that there is a predecessor cycle composed exclusively of nodes of Ω. All nodes of Ω have a predecessor and this predecessor is in Ω. Since Ω is finite, there is at least one node in Ω that is an indirect predecessor of itself.

Since we have a predecessor cycle, we can safely remove an edge of this cycle without altering the problem. This means that one fluent f in Ω is deprived of one of its achiever. f gets a new predecessor and its earliest appearance is updated. Note that the new $pred(f)$ is still in Ω by definition of a self-dependent set.

In this new model, Ω is still a self-dependent set. The above steps can thus be repeated until one fluent of Ω has no achievers left. This fluent and all actions depending on it can be deleted. The nodes that are left from Ω still form a self-dependent set, the above procedure can thus be repeated until all fluents and actions of Ω have been proved unfeasible. □

Example A.1. The graph in Figure A.1 shows a problem with no possible actions and fluents. We display a possible combination of predecessor edges (in red) to highlight the presence of a self-dependent set. One self-dependent set in this problem is $\{b, A_1, A_2, a, B\}$.

We have a cycle of predecessors A_1aBb of with an accumulated delay (sum of the labels) of 1. This cycle can be read as *"If A_1 is the first achiever of a and B is the first achiever of b then a can only be achieved at time t if it was achieved at time $t-1$."* This is of course not possible: either b or a needs another first achiever. The only possibility is to select A_2 as first achiever for a and a can be removed from the effects of A_1.

In this equivalent model, A_2 is the new predecessor of a which results in a predecessors cycle A_2aBb. Consequently, achieving a and b require selecting another first achiever for one of them. Since we have no other options left, all nodes in this cycle are not possible.

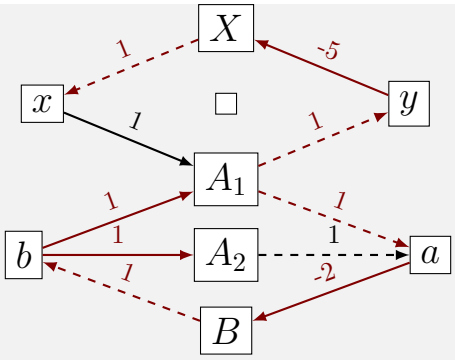


Figure A.1: Problem in graph representation. Edges in red represent a possible assignment of predecessors at some point in propagation.

A.2.2 From propagation to the identification of self-dependent set

We have identified sufficient conditions to declare a group of node impossible. We now show how the identification of such set can be integrated in Algorithm 2.

We say that a set of nodes $\mathcal{L}$ is late if any node in $\mathcal{L}$ has an earliest appearance at least d_{max} time units greater than any node not in $\mathcal{L}$; d_{max} being the maximum delay of any edge of the graph.

$$\forall x \notin \mathcal{L}, y \in \mathcal{L}, ea(x) + d_{max} < ea(y) \tag{A.1}$$

The intuition behind the definition of a late set is that all late nodes are separated from non-late nodes by a temporal gap. Furthermore, this temporal gap is big enough so that the earliest appearances of late nodes could not have been influenced by a non-late node (i.e. the predecessor of a late node is a late node).

Proposition A.2.4. *If $\mathcal{L}$ is a set of late nodes, then $\mathcal{L}$ is a self-dependent set.*

Proof. We prove the two conditions of a set to be self-dependent.

Lets call Y the current predecessor of a node X and e_{YX} the edge from Y to X. At the last update of X, $ea(X)$ was set to $ea(Y) + lbl(e)$. We know that $ea(Y)$ can only increase and that $lbl(e) \leq d_{max}$. Consequently, we still have $ea(Y) + d_{max} \geq ea(X)$ and Y is in $\mathcal{L}$. We have shown that if X is in $\mathcal{L}$ then $pred(X)$ is in $\mathcal{L}$.

Let us take x a fluent in $\mathcal{L}$. Because a fluent takes the minimum earliest appearance of all its achievers, no such achiever can be more than d_{max} time units before it. All achievers of a fluent in $\mathcal{L}$ are thus in $\mathcal{L}$ as well. □

Corollary 1. *Any node in a late set is not reachable.*

Proposition A.2.5. *Any non-reachable node will eventually be part of a late set.*

Proof. All earliest appearances of possible nodes converge towards a finite value. On the other hand, earliest appearances of non-possible nodes diverge towards $+\infty$. At some point, the earliest appearance of unreachable nodes will be greater by d_{max} than the latest possible node. □

A possible implementation of the DeleteInfeasibleNodes function from Algorithm 2 is thus to check whether a late set of late events appeared during the propagation and delete all those. Such an implementation is guaranteed to finish and remove all unreachable nodes.

Appendix B

ANML domains

B.1 Blocks-PartHier

```
type Location;
type Block < Location;

predicate clear(Block b);
predicate handempty();
function Location on(Block b);

instance Location TABLE, HAND;


action pickup(Block b) {
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == TABLE :-> HAND;
  [all] handempty == true :-> false;
};

action putdown(Block b) {
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == HAND :-> TABLE;
  [all] handempty == false :-> true;
};

action stack(Block b, Block c) {
  motivated;
  duration := 5;
  [all] on(b) == HAND :-> c;
  [all] handempty == false :-> true;
  [all] clear(c) == true :-> false;
  [all] clear(b) == true;
};

action unstack(Block b, Block c) {
  duration := 5;
  [all] on(b) == c :-> HAND;
  [all] handempty == true :-> false;
  [all] clear(b) == true;
  [all] clear(c) == false :-> true;
};

action DoStack(Block a, Block b) {
  motivated;
  :decomposition {
    [all] on(a) == b;
  };
  :decomposition {
    [all] stack(a,b);
  };
};
```

B.2 Blocks-FullHier

```
type Location;
type Block < Location;

predicate clear(Block b);
predicate handempty();
function Location on(Block b);

instance Location TABLE, HAND;


action pickup(Block b) {
  motivated;
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == TABLE :-> HAND;
  [all] handempty == true :-> false;
};

action putdown(Block b) {
  motivated;
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == HAND :-> TABLE;
  [all] handempty == false :-> true;
};

action stack(Block b, Block c) {
  motivated;
  duration := 5;
  [all] on(b) == HAND :-> c;
  [all] handempty == false :-> true;
  [all] clear(c) == true :-> false;
  [all] clear(b) == true;
};

action unstack(Block b, Block c) {
  motivated;
  duration := 5;
  [all] on(b) == c :-> HAND;
  [all] handempty == true :-> false;
  [all] clear(b) == true;
  [all] clear(c) == false :-> true;
};

action uncover(Block a) {
  motivated;
  :decomposition {
    [all] clear(a) == true;
  };
  :decomposition {
    [start] clear(a) == false;
    constant Block onA;
    [start] on(onA) == a;
    [all] ordered(
      uncover(onA),
      unstack(onA,a),
      putdown(onA));
  };
};

action DoStack(Block a, Block b) {
  motivated;
  :decomposition {
    [all] on(a) == b;
  };
  :decomposition {
    [start] on(a) == TABLE;
    [all] ordered(
      uncover(a),
      uncover(b),
      p: pickup(a),
      s: stack(a, b));
      end(p) = start(s);
```

```
  };
  :decomposition {
    constant Block other;
    other != TABLE;
    [start] on(a) == other;
    [all] ordered(
      uncover(a),
      uncover(b),
      u: unstack(a, other),
      s: stack(a, b));
    end(u) = start(s);

  };
};
```

B.3 RACE

```
type Area with {
  constant boolean isManipulationArea;
  constant PreArea pre_area;
  constant ManArea man_area;
};
type PlArea < Area;  // area to place objects
type ManArea < Area; // area from which the robot manipulates
type PreArea < Area; // area from which to access a manipulation area
type NavArea < Area; // other area where the robot can be

// this would be useless with a primitive to check the type of a variable in ANML
forall(ManArea a) { a.isManipulationArea := true; };
forall((PlArea or PreArea or NavArea) b) { b.isManipulationArea := false; };

type Robot with {
  constant Arm left_arm;
  constant Arm right_arm;
  fluent Area loc; // location of the robot
  fluent TorsoPosture torso; // posture of the torso
  fluent boolean busy; // the robot is working on an exclusive task
};
type Arm with {
  fluent ArmPosture posture;
  fluent boolean free;
  constant Robot owner;
};

type Object with {
  fluent (PlArea or Arm) loc; // an object is either on a place area or in a robot's arm
};
type CoffeeJug < Object;
type MilkPot < Object;
type SugarPot < Object;

type ArmPosture;
instance ArmPosture ArmTuckedPosture, ArmUntuckedPosture, ArmUnnamedPosture,
    ArmToSidePosture, ArmCarryPosture;

type TorsoPosture;
instance TorsoPosture TorsoDownPosture, TorsoMiddlePosture, TorsoUpPosture;

// an object on 'pl' can be manipulated from 'man' and 'man' can be accessed from 'pre'
constant boolean connected(PlArea pl, ManArea man, PreArea pre);



/*** Primitive actions ***/

action move_base(Robot r, Area to) {
  motivated;
  end >= start+5000;
  constant Area from;
```

```
  [all] r.loc == from :-> to;
};

// only applicable between a manipulation area and its pre-area
action move_base_blind(Robot r, Area to) {
  motivated;
  end >= start+4000;

  constant ManArea man;

  constant Area from;
  from in { man, man.pre_area };
  to in { man, man.pre_area };
  from != to;

  [all] r.loc == from :-> to;
};

action tuck_arms(Robot r, ArmPosture left_pos, ArmPosture right_pos) {
  motivated;
  left_pos in { ArmTuckedPosture, ArmUntuckedPosture };
  right_pos in { ArmTuckedPosture, ArmUntuckedPosture };
  end >= start +4000;
  constant ArmPosture rarm_ini_pos;
  constant ArmPosture larm_ini_pos;
  [all] r.left_arm.posture == larm_ini_pos :-> left_pos;
  [all] r.right_arm.posture == rarm_ini_pos :-> right_pos;
};

action move_torso(Robot r, TorsoPosture new) {
  motivated;
  end >= start +4000;
  constant TorsoPosture old;
  [all] r.torso == old :-> new;
};

action pick_up_object(Robot r, Arm arm, Object obj) {
  motivated;
  end >= start +4000;
  r == arm.owner;
  constant PlArea pl;
  constant ManArea man;
  connected(pl, man, man.pre_area);
  [all] obj.loc == pl :-> arm;
  [all] r.loc == man;
  [all] arm.free == true :-> false;
};

action place_object(Robot r, Arm arm, Object obj, PlArea pl) {
  motivated;
  r == arm.owner;
  end >= start +4000;
  constant ManArea man;
  connected(pl, man, man.pre_area);
  [start] arm.posture == ArmToSidePosture;
  [all] r.loc == man;
  [all] obj.loc == arm :-> pl;
  [all] arm.free == false :-> true;
};

action move_arm_to_side(Robot r, Arm arm) {
  motivated;
  arm.owner == r;
  end >= start +4000;
  constant ArmPosture old_posture;
  [all] arm.posture == old_posture :-> ArmToSidePosture;

  // at least one of left/right arms must be untucked at start
  constant Arm any_arm;
  any_arm in { r.left_arm, r.right_arm };
  [start] any_arm.posture != ArmTuckedPosture;
};
```

```
action move_arms_to_carryposture(Robot r) {
  motivated;
  end >= start +4000;
  constant ArmPosture prevLeft;
  constant ArmPosture prevRight;
  [all] r.left_arm.posture == prevLeft :-> ArmCarryPosture;
  [all] r.right_arm.posture == prevRight :-> ArmCarryPosture;
  [all] r.torso != TorsoDownPosture;
};

action observe_objects_on_area(Robot r, PlArea pl) {
  motivated;
  end >= start +4000;
  constant ManArea man;
  connected(pl, man, man.pre_area);
  [all] r.loc == man;
};



/**** High Level Actions ****/

action adapt_torso(Robot r, TorsoPosture pose) {
  motivated;
  :decomposition {
    [all] r.torso == pose;
  };
  :decomposition {
    [start] r.torso != pose;
    [all] move_torso(r, pose);
  };
};

action torso_assume_driving_pose(Robot r) {
  motivated;
  :decomposition { // both arms are free
    [all] r.left_arm.free == true;
    [all] r.right_arm.free == true;
    [all] adapt_torso(r, TorsoDownPosture);
  };
  :decomposition { // there is an occupied arm
    constant Arm arm;
    arm.owner == r;
    [all] arm.free == false;
    [all] adapt_torso(r, TorsoMiddlePosture);
  };
};

action adapt_arms(Robot r, ArmPosture pose) {
  motivated;
  :decomposition {
    [all] r.left_arm.posture == pose;
    [all] r.right_arm.posture == pose;
  };
  :decomposition { // we want arms to be tucked
    constant Arm arm;
    pose == ArmTuckedPosture;
    arm.owner == r;
    [start] arm.posture != ArmTuckedPosture;
    [all] tuck_arms(r, ArmTuckedPosture, ArmTuckedPosture);
  };
  :decomposition { // we want arms in carry posture
    pose == ArmCarryPosture;
    constant Arm arm;
    arm.owner == r;
    [start] arm.posture != ArmCarryPosture;
    [all] move_arms_to_carryposture(r);
  };
};

action arms_assume_driving_pose(Robot r) {
  motivated;
  :decomposition { // both arms are free
```

```
    [all] r.left_arm.free == true;
    [all] r.right_arm.free == true;
    [all] adapt_arms(r, ArmTuckedPosture);
  };
  :decomposition { // there is an occupied arm
    constant Arm arm;
    arm.owner == r;
    [all] arm.free == false;
    [all] adapt_arms(r, ArmCarryPosture);
  };
};


action drive_robot(Robot r, Area to) {
  motivated;
  :decomposition {
    [all] r.loc == to;
  };
  :decomposition { // not in manipulation area, assume driving pose and drive
    constant Area from;
    from != to;
    from.isManipulationArea == false;
    [start] r.loc == from;
    [all] contains ordered(
      unordered(
        torso_assume_driving_pose(r),
  arms_assume_driving_pose(r)),
      move_base(r,to));
  };
  :decomposition { // in manipulation area, first go to matching pre-manip area
    constant ManArea from;
    from != to;
    from.isManipulationArea == true;
    [start] r.loc == from;
    [all] contains ordered(
      move_base_blind(r, from.pre_area),
      unordered(
        torso_assume_driving_pose(r),
  arms_assume_driving_pose(r)),
      move_base(r,to));
  };
};

action arm_to_side(Robot r, Arm arm) {
  motivated;
  arm.owner == r;
  :decomposition {
    [all] arm.posture == ArmToSidePosture;
  };
  :decomposition {
    [start] arm.posture != ArmToSidePosture;
    [all] move_arm_to_side(r, arm);
  };
};

action move_both_arms_to_side(Robot r) {
  motivated;
  :decomposition {
    [start] r.left_arm.posture == ArmTuckedPosture;
    [start] r.right_arm.posture == ArmTuckedPosture;
    [all] contains ordered(
      tuck_arms(r, ArmUntuckedPosture, ArmUntuckedPosture),
      unordered(
        arm_to_side(r, r.left_arm),
  arm_to_side(r, r.right_arm)));
  };
  :decomposition {
    constant Arm a;
    a in { r.right_arm, r.left_arm };
    [start] a.posture != ArmTuckedPosture;
    [all] contains ordered( // to stick to the original domain, should probably be
    unordered in practice
      arm_to_side(r, r.left_arm),
```

```
      arm_to_side(r, r.right_arm));
  };
};

action assume_manipulation_pose(Robot r, ManArea l) {
  motivated;
  :decomposition { // already in position
    [all] {
      r.left_arm.posture == ArmToSidePosture;
      r.right_arm.posture == ArmToSidePosture;
      r.loc == l;
    };
    [all] adapt_torso(r, TorsoUpPosture);
  };
  :decomposition { // at pre-area, assume pose and go to manipulation area
    [start] r.loc == l.pre_area;
    [all] contains ordered(
      unordered(
        adapt_torso(r, TorsoUpPosture),
  move_both_arms_to_side(r)),
      move_base_blind(r, l));
  };
  :decomposition { // go back to pre-area before assuming pose
    [start] r.loc == l;
    constant Arm any_arm;
    any_arm in { r.left_arm, r.right_arm };
    [start] any_arm.posture != ArmToSidePosture;

    [all] contains ordered(
      move_base_blind(r, l.pre_area),
      unordered(
        adapt_torso(r, TorsoUpPosture),
  move_both_arms_to_side(r)),
      move_base_blind(r, l));
  };
};

action leave_manipulation_pose(Robot r, ManArea l) {
  motivated;
  :decomposition {
    [start] r.loc == l;
    [all] move_base_blind(r, l.pre_area);
  };
};


action get_object(Robot r, Object o) {
  motivated;

  constant ManArea man;
  constant PlArea pl;
  connected(pl, man, man.pre_area);
  [start] o.loc == pl;
  constant Arm arm;
  arm in { r.left_arm, r.right_arm };

  :decomposition {
    [start] r.loc == man.pre_area; // this is not checked in their version (because
    leading to much backtracking?)
    [all] contains ordered(
      assume_manipulation_pose(r, man),
      pick_up_object(r, arm, o));
  };
  :decomposition {
    [start] r.loc != man.pre_area;
    [all] contains ordered(
      drive_robot(r, man.pre_area),
      assume_manipulation_pose(r, man),
      pick_up_object(r, arm, o));
  };
};

action put_object(Robot r, Object o, PlArea pl) {
```

```
  motivated;
  constant ManArea man;
  connected(pl, man, man.pre_area);

  constant Area initialLoc;
  [start] r.loc == initialLoc;

  constant Arm arm;
  [start] o.loc == arm;

  :decomposition {
    initialLoc != man;
    initialLoc != man.pre_area;
    [all] ordered(
      drive_robot(r, man.pre_area),
      assume_manipulation_pose(r, man),
      place_object(r, arm, o, pl));
  };
  :decomposition {
    initialLoc == man.pre_area;
    [all] ordered(
      assume_manipulation_pose(r, man),
      place_object(r, arm, o, pl));
  };
  :decomposition {
    initialLoc == man;
    [all] place_object(r, arm, o, pl);
  };
};


// missing method to do nothing if there is nothing to do
action move_object(Robot r, Object o, PlArea area) {
  motivated;
  :decomposition {
    [all] ordered(
      get: get_object(r, o),
      put: put_object(r, o, area));

    // r cannot perform another get or put task concurrrently
    [start(get),end(get)] r.busy == false :-> false;
    [start(put),end(put)] r.busy == false :-> false;
  };
};
```

B.4 Cooking Dinner (POSTNU)

```
type Person;
instance Person wife, me;

type Location;
instance Location work, car, home, kitchen;

fluent boolean dinner_ready;
predicate should_notify_when_leaving(Person notified, Person notifier, Location l);
fluent Location loc(Person a);

/** Action to cook dinner, with uncontrollable duration between 25 and 30 minutes */
action cook(Person p) {
  duration :in [25,30];
  [all] loc(p) == kitchen;
  [end] dinner_ready := true;
  ::(observation_conditions(end) { /* none: always observable */ })
};

/** Ask "receiver" to text "sender" when leaving locaion "l" */
action text_me_when_you_leave(Person sender, Person receiver, Location l) {
  duration := 1;
  sender != receiver;
```

```
  [all] loc(receiver) == l;
  [end] should_notify_when_leaving(sender,receiver,l) := true;
};


/*** (Uncontrollable) Schedule of Wife ****/

// start ==[10,20]==> wife_working
wife_working :in start+ [10,20];

// wife_working ==[30,60]==> wife_driving
wife_driving :in wife_working + [30,60];

// wife_driving ==[35,40]==> wife_home
wife_home :in wife_driving + [35,40];

[wife_working] loc(wife) := work;
[wife_driving] loc(wife) := car;
[wife_home] loc(wife) := home;


::(observation_conditions(wife_driving) {
  [wife_driving] should_notify_when_leaving(me,wife,work) == true;
})
::(observation_conditions(wife_home) { /* none: always observable */ })


/*** Initial state ***/

[start] loc(me) := kitchen;
[start] dinner_ready := false;


/*** Goal: dinner be ready at most 5 minutes after wife gets home
     and no earlier than 5 minutes before ***/

[t-1] dinner_ready == false;
[t] dinner_ready == true;
wife_home -5 <= t;
t <= wife_home+5;
```

B.5 Search and Transport

```
type Location;
type Surface < Location; // surface (e.g. table) where items can be
type RobotLocation < Location; // places where the robot can be

// denotes whether a surface can be reached from a given location
constant boolean reachable_from(Surface iLoc, RobotLocation rLoc);

// respectively denote the minimal and maximal durations to go from one place to another
constant integer min_travel_dur(RobotLocation from, RobotLocation to);
constant integer max_travel_dur(RobotLocation from, RobotLocation to);

type Robot with {
  constant Gripper left_gripper;
  constant Gripper right_gripper;
  constant Head head;
  fluent RobotLocation loc;
};

type Gripper with {
  fluent boolean free;
};

type Head with { // Head of a robot whose camera are used for scanning surfaces
  fluent boolean used;
};
```

```
type Item with {
  fluent (Surface or Gripper) loc;
};

fluent boolean path_planner_free;



/** Action for going to a given location.
    The action is decomposed in two phases.
    The first one [start,t_have_traj] is the phase dedicated to planning a trajectory
    between the two locations.
    The second one [t_exec_traj,end] is the phase where the trajectory previously
    planned is executed. */
action Go(Robot r, RobotLocation to) {
  constant RobotLocation from; // location of the robot before executing its trajectory

  // contingent duration of the first phase (typically fast for path planning)
  // start ==[1,4]==> t_have_traj
  t_have_traj :in start + [1,4];

  // contingent duration of executing a (yet unknown) trajectory
  // t_exec_traj ==[min,max]==> end
  end :in t_exec_traj + [min_travel_dur(from,to),max_travel_dur(from,to)];

  // require the path planner to be free during the first phase
  [start,t_have_plan] path_planner_free == true :-> true;

  // denote the effects of executing the trajectory during the second phase: the robot
    will go from its initial location (from) to its target location (to)
  [t_exec_traj,end] r.loc == from :-> to;

  // force the planner not to compute the trajectory to early
  t_exec_traj <= t_have_traj +10;
};



action SearchTransport(Robot r, Item i, Surface table) {
  duration :in [30, 300]; // large uncertainty regarding the duration

  /** make sure nobody uses the robot but state that it will be in a location from
    which "table" can be reached at the end */
  constant RobotLocation robot_start;
  constant RobotLocation robot_end;
  reachable_from(table,robot_end);
  [all] r.loc == robot_start :-> robot_end;

  // make sure the robot has at least one free gripper
  constant Gripper g;
  g in { r.left_gripper, r.right_gripper };
  [all] g.free == true :-> true; // gripper must be free at start and will be free at
    end

  // make sure nobody uses the head as it will be needed for scanning surfaces and
    perform pick/place actions
  [all] r.head.used == false :-> false;

  // location of "i" is initially unknown, but it will be on the table at the end
  [start] i.loc == UNKOWN;
  [end] i.loc := table;
};



action Transport(Robot r, Item i, Surface table) {
  [start] i.loc != UNKNOWN;
  [all] contains ordered(
     GoPick(r,i), GoPlace(r,i,table);
  };
};
```

```
action GoScan() {
  // "r" must be in a location from which "s" is visible
  constant RobotLocation l;
  visible_from(s,l);
  :decomposition{
    [all] r.loc == l;
    [all] contains Scan(r,s);
  };
  :decomposition {
    [start] r.loc != l;
    [all] contains ordered(
      Go(r,l),
      Scan(r,s));
  };
};


action Scan(Robot r, Surface s) {
  duration :in [10,20];

  // "r" must be in a location from which "s" is visible
  constant RobotLocation l;
  visible_from(s,l);
  [all] r.loc == l;

  // nobody else must be using the head while scanning
  [all] r.head.used == false :-> false;

  [end] s.scanned := true;
};
```