HAL
open science

FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning

Arthur Bit-Monnot, Malik Ghallab, Félix Ingrand, David E Smith

To cite this version:

Arthur Bit-Monnot, Malik Ghallab, Félix Ingrand, David E Smith. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. 2020. hal-02987044

HAL Id: hal-02987044

https://laas.hal.science/hal-02987044v1

Preprint submitted on 3 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

arXiv:2010.13121v1 [cs.AI] 25 Oct 2020

FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning

Arthur Bit-Monnot[1], Malik Ghallab[1], Félix Ingrand[1], and David E. Smith[2]

[1]LAAS-CNRS, Université de Toulouse, INSA, CNRS, France
[2]Formerly at NASA Ames Research Center, USA

Abstract

Temporal planning offers numerous advantages when based on an expressive representation. Timelines have been known to provide the required expressiveness but at the cost of search efficiency. We propose here a temporal planner, called FAPE, which supports many of the expressive temporal features of the ANML modeling language without loosing efficiency.

FAPE's representation coherently integrates flexible timelines with hierarchical refinement methods that can provide efficient control knowledge. A novel reachability analysis technique is proposed and used to develop *causal networks* to constrain the search space. It is employed for the design of informed heuristics, inference methods and efficient search strategies. Experimental results on common benchmarks in the field permit to assess the components and search strategies of FAPE, and to compare it to IPC planners. The results show the proposed approach to be competitive with less expressive planners and often superior when hierarchical control knowledge is provided. FAPE, a freely available system, provides other features, not covered here, such as the integration of planning with acting, and the handling of sensing actions in partially observable environments.

Contents

1 Introduction **3**

2 A Hierarchical Time-oriented Representation **4**
2.1 Main Components . . . 4
2.2 Tasks and Action Templates . . . 7
2.3 Chronicles . . . 8
2.4 Plan: Transformations and Solutions . . . 10
2.5 Discussion . . . 12

3 A Plan-Space Planning Procedure **12**
3.1 Overview . . . 12
3.2 Flaws and Resolvers . . . 13
3.3 Constraint Networks . . . 18
3.4 Search Space: Properties and Exploration . . . 20

4 Search Control **22**
4.1 Instantiation and Refinement Variables . . . 22
4.2 Reachability Analysis . . . 24
4.3 Causal Network . . . 33
4.4 Search Strategies . . . 39

5 Empirical Evaluation **42**
5.1 Evaluation of the Different Components of the Planner . . . 42
5.2 Empirical Comparison with IPC Planners . . . 50

6 Related work and discussion **52**
6.1 PDDL Temporal Planners . . . 52
6.2 Hierarchical planners . . . 53
6.3 Timeline based Representations and Planners . . . 55

7 Conclusion **56**

References **58**

A Overlength Proofs **64**
A.1 Proof of Soundness and Completeness (Proposition 3.2) . . . 64
A.2 Proof of the convergence of earliest appearances (Proposition 4.1) . . . 65
A.3 Proof that late nodes are unreachable (Proposition 4.2) . . . 66

B Example of ANML domains **69**
B.1 Dock Worker . . . 69
B.2 Blocks-PartHier . . . 70
B.3 Blocks-FullHier . . . 70

1 Introduction

There are numerous advantages in making time a central entity in planning. An explicit time representation permit in particular:

- to model action duration;
- to model effects, conditions, and resources borrowed or consumed by an action at various moments along its span, as well as delayed effects;
- to handle goals with relative or absolute temporal constraints;
- to plan with respect to expected future exogenous events;
- to plan with actions that maintain a value while being executed, as opposed to just changing that value (e.g., tracking a target, keeping a spring latch in position);
- to handle the concurrency of actions that have interacting and joint effects; and
- to allow for flexible plan execution through dispatching and synchronization mechanisms.

Several success stories in the application of automated planning have illustrated these advantages, e.g., in logistics [Wil+08] or space planning [Jon+00].

Planning with explicit time relies either on a *state-space representation* or a *timeline representation*. The former uses temporally qualified durations between states, i.e., instantaneous snapshots of the entire system. The latter describes possible evolutions of individual state variables over time, i.e., partial local views of state trajectories, together with temporal constraints between elements of timelines.

Recent contributions to temporal planning have generally used a state-space representation based on PDDL2.1 [FL03]. This is explained by the wealth of search techniques, heuristics, and test domains that have been developed for state-space planning, resulting in significant performance improvements. However, these planners, but of a few exceptions, do not have all the abilities listed above, in particular the handling of required concurrency.

The timeline representation is more expressive, as it focuses on local changes and partial plans, which correspond to entire sets of states. A few timeline modeling languages have been proposed, e.g., IxTeT [GL94], AML [Rab+99], NDDL [FJ03], or the more expressive ANML [SFC08]. The timeline representation is often perceived as having an efficiency drawback compared to state-space planning: it mostly relies on plan-space search algorithms and CSP techniques, for which known heuristics are not as efficient as state-space heuristics. Only a few planners have been proposed for the timeline languages. In particular, to our knowledge, no planner but FAPE fully support the temporal or the hierarchial features of the powerful ANML language.

Our purpose in developing FAPE (*Flexible Acting and Planning Environment*) was mainly to benefit from the expressivity of a representation combining timelines and task hierarchy. We developed a reachability analysis and heuristics which make our timeline approach competitive. We specifically report here on the following contributions:

1. We introduce a temporal planning representation, which consistently blends timelines with hierarchical refinement methods to allow for additional control knowledges;
2. We present a planning algorithm for the proposed representation, analyze its search space and prove its soundness and completeness;
3. We propose an original reachability analysis method for a relaxed problem integrating earliest times; we use it to develop specific causal networks and potential causal chains toward goals;

we exploit the preceding structures for the design of informed heuristics;

4. We propose search strategies that take advantage of the designed heuristics and the hierarchical control knowledge, when available;
5. We present empirical evaluation results on classical benchmarks of the field, which allows us to assess the various components and search strategies of FAPE, and to compare it to IPC planners; the results show that FAPE is competitive with less expressive planners, and generally superior when hierarchical control knowledge is introduced.

Other features of FAPE are not covered here for the sake of focus, e.g., the integration with acting, or observation actions in partially observable environments [BMGI16; BM16].[1]

The following five sections detail successively the five preceding items. Related work is discussed in Section 6, before a conclusion.

2 A Hierarchical Time-oriented Representation

Our representation is based on the ANML language [SFC08]. We use the same notion of actions whose conditions and effects are stated as assertions on multi-valued state variables. Assertions are related to an action's start and end timepoints through temporal constraints and can appear at arbitrary times. As in ANML, our representation allows for describing the expected evolution of the environment and temporally extended goals.

A subtle difference with ANML lies in the encoding of hierarchical knowledge. A high-level action in ANML has subactions expressed through additional assertions on dedicated state variables. Instead, we provide a separate notion of *task* which relates to the HTN planning. This is complemented by an optionnal notion of *task-dependency*, encoding the fact that an action must be part of a task network as in a classical HTN. Task-dependency enables to handle generative planning as well as HTN planning domains in a uniform way.

Example 2.1 (Running Example). For the purpose of illustrating the representation, we use a restricted version of the *dock worker* domain of [GNT04]. There, a harbor with several connected *docks* is served by automated *trucks* and *cranes* that move containers between different locations to load and unload ships. We have three primitive actions:

- $\mathsf{Move}(r, d, d')$: truck r moves from dock d to dock d' if the two are `connected`. A dock can contain a single truck. The duration depends on the distance between d and d'.
- $\mathsf{Load}(r, c, d)$: container c is loaded onto a truck r if both are in the same dock d where a crane is available to perform the loading.
- $\mathsf{Unload}(r, c, d)$: container c is unloaded from a truck r in a dock d if c is currently on r and r is at dock d where a crane is available to perform the task.

We are interested in planning the loading and unloading of ships that will be docked at a specific location for a limited amount of time. □

2.1 Main Components

Temporal Variables and Constraints. We use a quantitative time representation based on timepoints. We rely on temporal variables (e.g. t, t_1), designating a timepoint and ranging over

[1] FAPE and the tested domains are publicly available at: `https://github.com/laas/fape`

integers. Temporal variables are constrained through the usual arithmetic operators 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 attached to specific events such as the start of an action or the instant at which a given condition must be fulfilled.

Atemporal Variables and Constraints. We consider a finite set of domain constants O, e.g., docks or trucks in Example 2.1. A *type* as a subset of O, e.g., $\texttt{Docks} = \{\texttt{dock1}, \texttt{dock2}, \texttt{dock3}\}$. A type can be composed from other types by union or intersection (e.g. $\texttt{Vehicles} = \texttt{Ships} \cup \texttt{Trucks}$).

An *object variable* o 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.[2]

A *constraint* over a set of variables $\{x_1, \ldots, x_n\}$ is a pair $(\langle x_1, \ldots, x_n \rangle, \gamma_R)$ where $\gamma_R \subseteq dom(x_1) \times \cdots \times dom(x_n)$ is the relation of the constraint, giving the allowed values for the tuple of variables. γ_R can be given as a table of allowed values or a function, e.g., $\textit{travel-time}(d, d') = \delta$ is met when δ is the time it takes from d to d'.

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

State variables and fluents. The state evolution over time is captured by multi-valued *state variables*. A state variable maps time and a tuple of objects to an object. For instance $\texttt{loc} : \texttt{Time} \times \texttt{Trucks} \rightarrow \texttt{Docks}$ gives the position of a truck 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 meaning $sv(t, x_1, \ldots, x_n) = v$. A complete definition of the state of the environment at a given time is specified by taking a snapshot of all state variables at that moment.

A *fluent* is a pair of a state variable sv and its value v, denoted $\langle sv{=}v \rangle$, and is said to hold at time t if sv has the value v at time t.

Temporally Qualified Assertions. An *assertion* is a temporally qualified fluent. We use *persistence* and *change* assertions to express knowledge or constraints on the evolution of a state variable (as in [GNT16, Sec. 4.2.1]).

A *persistence* assertion, denoted $\langle [t_s, t_e]\, sv{=}v \rangle$, requires the state variable sv to keep the same value v over the temporal interval $[t_s, t_e]$. In planning domains, persistence assertions are typically used to express requirements such as goals or pre-conditions of actions. For instance, the persistence assertion $\langle [400, 500]\ \texttt{loc}\,(\texttt{r1}) = \texttt{dock2} \rangle$ can represent the objective that the truck `r1` is at `dock2` at time 400 and stays there through time 500. We also allow a persistence assertion to be defined at an instant t (denoted $[t]$) rather than over an interval.

A *change* assertion, denoted $\langle [t_s, t_e]\, sv{:}v_1 \mapsto v_2 \rangle$, asserts that the state variable sv changes 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 is a consequence of an agent's activity. For an action, a change assertion represents a pre-condition that must be met (sv must have the value v_1 at time t_s) and an effect of the action (sv will have the value v_2 at time t_e). Over the temporal interval $]t_s, t_e[$ the value of sv is unspecified.

[2]Note that numeric variables are distinct from temporal variables, which can have infinite domains.

This allows to represent durative change on a state variable without explicitly specifying the details of the transition. For instance, the change assertion $\langle [100, 150]\ \mathit{loc}(\mathtt{r1}) : \mathtt{dock1} \mapsto \mathtt{dock2} \rangle$, means that the truck `r1` will move from `dock1` to `dock2` over the temporal interval $[100, 150]$. The details of its location over $]100, 150[$ are unspecified.

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

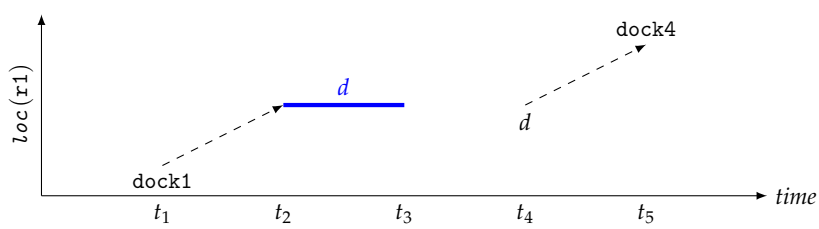
Assertions can involve object and temporal variables. A planner has to find activities to be performed and constraints on the variables such that the expected evolution of the system is both feasible and achieves the desired goals.

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 $\mathit{start}(\alpha)$ and $\mathit{end}(\alpha)$ respectively.

Timelines. 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 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 truck at different points in time.

Example 2.2. The following is a timeline containing three temporally qualified assertions on the state variable $\mathit{loc}(\mathtt{r1})$.

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



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

A timeline may have uninstantiated temporal and object variables: it represents a set of possible evolutions of a state variable. In Example 2.2, the state variable $\mathit{loc}(\mathtt{r1})$ will go through different values depending on the value assigned to d. Assertions related to an instantiated state variable, such as $\mathit{loc}(r)$ for $r \in \mathit{trucks}$, refer to a set of timelines.

A timeline $(\mathcal{F}, \mathcal{C}_1)$ is an instantiation of a timeline $(\mathcal{F}, \mathcal{C}_2)$ if *(i)* $\mathcal{C}_2 \subseteq \mathcal{C}_1$ and *(ii)* all variables in $\mathcal{F}$ and $\mathcal{C}_1$ are instantiated. While a state variable can only have a single value, it is possible for two assertions to require 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 $([t_1, t_2] \cap [t_3, t_4] \neq \varnothing) \wedge (v \neq v')$. These constraints need to be taken into account in a timeline to unsure its consistency.

Definition 2.1 (Possible Consistency of a Timeline)**.** 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 the state variable has no conflicting values in this instance.

Definition 2.2 (Necessary Consistency of a Timeline)**.** A timeline $(\mathcal{F}, \mathcal{C})$ is *necessarily consistent* if all its instantiations are consistent timelines.

A timeline that is not possibly consistent cannot describe a valid evolution of its state variable. A necessarily consistent timeline does not contain any conflicting assertions regardless of the choices made when binding its variables: it must have a set of constraints such that *(i)* no two change assertions can overlap, *(ii)* no persistence can overlap a change assertion, *(iii)* any two overlapping persistences must be of the same value (by definition of a timeline, they are on the same state variable). The timeline of Example 2.2 is necessarily consistent because the temporal constraints impede assertions from overlapping.

Definition 2.3 (Causal Support)**.** An assertion $\alpha \in \mathcal{F}$ is causally supported in a timeline $(\mathcal{F}, \mathcal{C})$, for $\alpha = \langle [t_1, t_2]\, sv{=}v \rangle$ or $\alpha = \langle [t_1, t_2]\, sv{:}v \mapsto v' \rangle$ iff:

- there is an assertion $\beta \in \mathcal{F}$ that produces $\langle sv{=}v \rangle$ at a time $t_0 \leq t_1$; β, called the causal supporter of α, is either an assignment $\langle [t_0]\, sv{:=}v \rangle$ or if it is a change $\langle [t, t_0]\, sv{:}v'' \mapsto v \rangle$; and
- sv keeps the value v from the end of β until the start of α, i.e., $\langle sv{=}v \rangle$ holds in $[t_0, t_1]$.

Any assignment assertion $\langle [t]\, sv{:=}v \rangle$ is *a priori* causally supported. □

In Example 2.2, the persistence assertion, $[t_2, t_3]\ \mathtt{loc}(\mathtt{r1}){=}d$ is causally supported by the first change assertion, $[t_1, t_2]\ \mathtt{loc}(\mathtt{r1}){:}\mathtt{dock1} \mapsto d$, since *(i)* their temporal intervals meet, and *(ii)* the fluent $\langle \mathtt{loc}(\mathtt{r1}){=}d \rangle$ produced is the one required. However none of the two change assertions are causally supported. This might seem surprising for the second change assertion because the fluent $\langle \mathtt{loc}(\mathtt{r1}){=}d \rangle$ is produced by the first assertion. This fluent is however not constrained to hold during $]t_3, t_4[$ and, as a consequence, $\mathtt{loc}(\mathtt{r1})$ might still be modified during that 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 way of supporting it would be the introduction of a new change assertion $\langle [t, t_4]\, sv{:}d' \mapsto d \rangle$, such an assertion would itself need to be causally supported.

To have all assertions of a timeline causally supported, the earliest one must be an *assignment*, which is by definition *a priori* supported.

2.2 Tasks and Action Templates

An action has a unique name and a set of parameters. The pre-conditions and 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 parameters and temporal variables in the action. We augment this usual view of an action with hierarchical features: *(i)* we define a set of parametrized task symbols $\mathcal{T}$, *(ii)* each action is associated with a task symbol in $\mathcal{T}$ representing the task this action achieves, *(iii)* an 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 used freely by the planner but only as a subtask of some other action.

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

- *head*(A) is the name and the list of typed parameters of A. The parameters refer to object variables. Temporal variables are kept implicit in the parameter list; t_{start} and t_{end} are respectively the start and end timepoints of A. An action template may also use a *duration variable* which is not a binding argument nor a free variable, but a mean to couple temporal and object constraint networks (see Section 3.3.3).
- *task*$(A) \in \mathcal{T}$ is the task it achieves with parameters from those of A.
- *dependent* $\in \{\top, \bot\}$ is true if the action is *task dependent*, otherwise the action is said to be *free*. The 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.[3]
- *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 $task(a) = \tau$. The timepoints t_s and t_e need not be grounded; they are related to other timepoints through temporal constraints, possibly inducing a partial order on the subtasks.
- $(\mathcal{F}_A, \mathcal{C}_A)$ is a set of possibly consistent timelines. $\mathcal{F}_A$ provides the conditions and effects of A on some state variables through persistence and change assertions. $\mathcal{C}_A$ imposes constraints on the possible instantiations of the temporal and object variables of A.

In hierarchical planning, one often distinguishes between high-level actions (or abstract tasks) and primitive actions (or primitive tasks). A *high-level action* provides a *method* to achieve a given task, expressed as a set of conditions and subtasks. A *primitive action* is intended to be executed with a command whose effects are represented as assertions on state variables. While our model does not make a distinction between the two, we will use the same terminology when needed to convey the role of an action in a planning model.

Figure 1 illustrates a primitive action template for moving a robot r from dock d to d'. It requires r to be in d at t_{start}, and d' to be empty at some point t' before t_{end}. Its effects are to make the location of the truck be d' at t_{end}, and to have d empty at some point t after t_{start} and before t'. The action has no subtask and is the only achiever of the eponymous task `move`(r, d, d'). Figure 2 exemplifies high-level actions offering two methods for achieving the `transport` task of moving a container c to a location d.

2.3 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. It extends the existing notion of chronicle [GNT16, Sec. 4.2.4] with a new member π that keep tracks of tasks and actions in the plan.

[3]The current draft 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.

move(r, d, d')	
task:	$\texttt{move}(r, d, d')$
dependent:	no
subtasks:	$\varnothing$
assertions:	$[t_{start}, t_{end}]\ \texttt{loc}(r) : d \mapsto d'$
	$[t_{start}, t]\ \texttt{occupant}(d) : r \mapsto \texttt{nil}$
	$[t', t_{end}]\ \texttt{occupant}(d') : \texttt{nil} \mapsto r$
constraints:	$\mathit{connected}(d, d') = \top$
	$t_{end} - t_{start} = \mathit{travel\text{-}time}(d, d')$
	$t < t'$

Figure 1: An action to move a truck r from dock d to dock d'. The duration of the action $(t_{end} - t_{start})$ is set to be equal to the travel time from d to d'. $\mathit{connected}(d, d') = \top$ is a constraint on objects variables and requires that the two docks be connected, a temporally invariant property of the planning domain. Left implicit for readability are the types of variables and the constraints $t_{start} < t_{end}, t_{start} < t$ and $t' < t_{end}$.

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

m2-transport(r, c, d_s, d)	
task:	$\texttt{transport}(c, d)$
dependent:	yes
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:	$\texttt{connected}(d_s, d)$
	$d_s \neq d$
	$t_1 < t_2 < t_3 < t_4$

Figure 2: High-level actions for achieving the task of transporting a container c to a location d. The first one 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 truck r.

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

- π_0 a set of temporally qualified tasks that must be achieved.
- $\mathcal{F}_0$ a set of temporally qualified assertions that describe the initial state of the environment and its expected evolution together with a set of goals. The initial state and its evolution is typically depicted by *assignment assertions* (that do not require causal support). Goals are typically represented by *persistence assertions* whose causal support will require additional activity to be considered by the planner.
- $\mathcal{C}_0$ is a set of constraints restricting the allowed values for the temporal and object variables in π_0 and $\mathcal{F}_0$. They express temporally extended goals, timed initial literals and ordering constraints on the tasks to be achieved.

The chronicle below represents a planning problem with two trucks `r1` and `r2`, initially at `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 container `c1` that is initially on `ship1` to `dock3` and to have the two trucks in their initial locations at the end. Note that ϕ_0 states the planning objectives *(i)* as tasks to perform, as in HTN, and *(ii)* as goals to satisfy, as in generative planning.

ϕ_0	
tasks:	$[t, t']$ `transport(c1,dock3)`
assertions:	$[t_{start}]$ loc(`r1`)$:=$`dock1`
	$[t_{start}]$ loc(`r2`)$:=$`dock2`
	$[t_{start}]$ on(`c1`)$:=$`ship1`
	$[t_{start} + 10]$ $docked$(`ship1`)$:=$`pier1`
	$[t_{start} + \delta]$ $docked$(`ship1`)$:=$ nil
	$[t_{end}]$ loc(`r1`)$=$`dock1`
	$[t_{end}]$ loc(`r2`)$=$`dock2`
constraints:	$t_{start} < t < t' < t_{end} \wedge 20 \leq \delta \leq 30$

Figure 3: The initial chronicle of a planning problem.

Planning with this representation is a sequence of transformations of a chronicle, starting from ϕ_0, until a solution plan is found. We now describe the possible transformations and the conditions a solution plan must meet.

2.4 Plan: Transformations and Solutions

HTN planners build plans by systematically decomposing tasks, while generative planers synthesizes them by iteratively introducing new action instances. In FAPE, both processes are combined, interleaved with the insertion of temporal and binding constraints.

2.4.1 Task Refinement

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 such that $task(a) = \tau$. This transformation

is denoted by $\phi \xrightarrow{\tau,a}_D \phi'$ and results in the following ϕ':

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

The refining action a may introduce additional tasks and assertions representing both the conditions and the effects of this action. In addition to the constraints from the action template, the constraint $task(a) = \tau$ 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.

2.4.2 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* action instance a, i.e., a is not task-dependent. This transformation is denoted by $\phi \xrightarrow{a}_I \phi'$ and results in the following ϕ':

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

Note that task dependent actions can only be used through task decomposition and must respect all constraints specified on that task.

2.4.3 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 restrictions $(\mathcal{F}, \mathcal{C})$, where $\mathcal{F}$ is a set of persistence assertions and $\mathcal{C}$ is a set of constraints over temporal and object variables. This transformation is denoted by $\phi \xrightarrow{(\mathcal{F},\mathcal{C})}_R \phi'$ and results in the following ϕ':

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

This transformation restricts a chronicle by either *(i)* adding persistence conditions to achieve the causal support of an assertion, (as for causal links in plan-space planning), or *(ii)* restricting allowed values of some variables to remove potential inconsistencies in the chronicle, for instance by imposing an order on two actions with conflicting requirements, or to unify task and actions instances that can be unified.

2.4.4 Reachable and Solution Plans

A chronicle ϕ' is *reachable* from a chronicle ϕ if there is a sequence of task refinements, action insertions, and plan restrictions that transformes ϕ into ϕ'. The planner has to find a sequence of transformations such that the resulting chronicle corresponds to a feasible plan.

Definition 2.4 (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 necessarily consistent set of timelines. □

The solution plan is given by the actions in π^* with the constraints from $\mathcal{C}^*$. Since ϕ^* must be reachable from ϕ_0, the actions in ϕ^* must fulfill hierarchical constraints: if an action in π^* is *task-dependent* then it was inserted through task refinement and must respect all constraints placed on the task it refines. Furthermore, all assertions in ϕ^* are causally supported. Finally, all instantiations of variables in π^* are consistent. Uninstantiated variables allow to delay choices until execution time. This is typically the case for temporal variables whose instantiation is often decided dynamically at execution time.

2.5 Discussion

The proposed representation builds on the rich temporal semantics of ANML. It 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. It also supports a unique mix of generative and hierarchical planning. The notion of task-dependency allows for a seamless integration of generative and hierarchical models especially allowing for partial hierarchies and the capability of allowing task insertion.

This representation extends the expressivity of existing planners allowing to encode generative planning problems (i.e. with no tasks nor task-dependent actions) as well as HTN problems (i.e. where all actions are task-dependent). In the following sections, we introduce a planning algorithm supporting this representation as well as several dedicated search control methods.

3 A Plan-Space Planning Procedure

3.1 Overview

FAPE extends the chronicle planning approach [GNT16, Sec. 4.3], which was initially proposed in IxTeT [GL94]. Our contribution is a planning procedure that seamlessly supports both hierarchical and generative planning.

Given a planning problem with an initial chronicle ϕ_0, the planner attempts to transform ϕ_0 into a solution chronicle ϕ^* where all tasks have been refined, all assertions are causally supported and the set of temporal assertions defines necessarily consistent timelines. For that, the planner detects *flaws* in a chronicle: features that prevent the chronicle from being a solution. A *flaw* in a chronicle $(\pi, \mathcal{F}, \mathcal{C})$ is either:

- an *unrefined task* $\tau \in \pi$,
- an *unsupported assertion* $\alpha \in \mathcal{F}$, or
- *conflicting assertions*: a pair of assertions $(\alpha, \beta) \in \mathcal{F} \times \mathcal{F}$ that can be conflicting given $\mathcal{C}$.

Each type of flaw matches one of the necessary conditions for a chronicle to be a solution plan (Definition 2.4); the last two types generalize respectively the notions of open-goals and threats

in plan-space planning. Every flaw must be resolved to transform a chronicle into a solution plan. Resolving a flaw requires the application of one or several plan transformations (Section 2.4).

Algorithm 1 is a nondeterministic abstract view of the planning procedure. For a given chronicle, the planner chooses a flaw φ to solve. Since all flaws must eventually be solved, this choice is not a backtracking point. Next, the planner nonderterministically chooses of a resolver ρ to handle the flaw; it may backtrack on this choice. The resolver results in a transformation of ϕ into a refined chronicle ϕ' in which the flaw φ is absent, and from which the search proceeds recursively. The algorithm returns when it encounters a solution plan (i.e. a flaw-less chronicle) or a deadend (i.e. a flaw with no resolvers).

Algorithm 1 FAPE algorithm: returns a solution plan achieving the tasks and goals in ϕ for the domain Σ, or failure if none exists.

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

This procedure follows the general refinement planning framework [KKY95; Sch09], which has been used by partial order planners [PW92; YS03], generative temporal planners [GL94; VG06], and hierarchical planners, e.g., HiPOP [Bec+14] and PANDA [Sch09]. The differences between all those planners lie in four aspects: *(i)* their definition of flaws and resolvers, *(ii)* the type of constraints and propagation used to reason about variables in the plan, *(iii)* their internal representation of a chronicle, and *(iv)* the strategy for exploring the search space

The remainder of this section details successively these first three aspects, while the search guidance will be addressed in Section 4.

3.2 Flaws and Resolvers

The definition of flaws is critical for the soundness of the planning procedure. Their resolvers condition the completeness. Ill-defined resolvers may also result in redundancies in the search space. Let consider the three types of flaws and their resolvers.

3.2.1 Unsupported Assertions

The first type of flaw is a lack of causal support for some assertion in the current chronicle. The planner incrementally tracks this property by associating every assertion that requires causal support to an assertion that produces the desired fluent. The causal link relation $\beta \rightarrow \alpha$ denotes that the assertion β is the causal supporter of α. A causal link is created by inserting an additional persistence assertion $[end(\beta), start(\alpha)]\, sv = v$, which prevents any change on the value of the state variable sv from the end of β until the start of α.

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. Any change or persistence assertion that does not have an incoming causal link constitutes an *unsupported assertion flaw*. Such a flaw in a chronicle ϕ is solved by finding another assertion β that can serve as a support: either β is already in $\mathcal{F}_\phi$, or β can be added through the insertion of an action.

Action insertion must take into account the hierarchical features of the domain. Instead of directly inserting an action that provides causal support, we select or create a refinement tree in which the supporting assertion will be chosen.

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

Possible effects represent fluents that can be produced as part of the refinement tree of a task or an action. A fluent f is a *possible effect* of an action a, denoted $f \in E_a^+$, if it is either a direct effect of a or a possible effect of the refinement of one of its subtasks. A fluent f is a *possible effect* of a task τ, denoted $f \in E_\tau^+$, if f is a possible effect of an action a that refines τ.

$$E_a^+ = \mathit{direct_effects}(a) \bigcup_{\tau \in \mathit{subtasks}(a)} E_\tau^+$$

$$E_\tau^+ = \bigcup_{m \,\in\, \{a \,\mid\, \mathit{task}(a)=\tau\}} E_m^+$$

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 1 has a direct effect $\langle \mathtt{loc}(r){=}d' \rangle$. It is produced at time t_{end}, thus its associated delay is the minimal duration of the action: $t_{end} - t_{start}$.

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

The fluent $\langle loc(r){=}d' \rangle$ is also a possible effect of the action m2-transport (Figure 2) because the presence of its `move` subtask means it can be produced as a result of inserting m2-transport in the plan. The delay associated with the possible effect is defined recursively as:

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

The computation of minimal delays is done by a shortest path computation in an STN containing all timepoints and temporal constraints of the action. □

A supporting assertion β for an unsupported assertion α is either:

- already in the plan,
- introduced by decomposing an unrefined task $\tau \in \pi$, such that τ has a possible effect supporting α, denoted $cond(\alpha) \in E_\tau^+$), or
- introduced through the addition of a *free* action a, such that a has a possible effect supporting α, i.e., $cond(\alpha) \in E_a^+$.

It is impractical for the planner to branch over all the assertions that can be used 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 commits to select its causal supporter in the set of assertions resulting from the future refinement of a particular unrefined task $\tau \in \pi_\phi$.

Supporting tasks commitments. When doing hierarchical planning, it might happen that several decomposition steps occur before a task is fully decomposed into primitive actions. On the contrary in plan-space planning, the resolution of an unsupported condition typically requires the immediate insertion of a supporting action in the plan. To reconcile the two approaches, we now present a mechanism that restricts the possible supporters of a condition to the actions descending from a particular task. Effectively this allows expressing search commitments of the form "a condition c must be supported by the yet unrefined task τ" and its corollary "τ must decomposed in a way that supports c". This construct is to be exploited by the resolvers that we present immediately after.

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 causal supporter of α can only be chosen in the possible effects of an action that is a descendant of τ. An action is a descendant of τ if it refines τ or if it 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 Supporter.** 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 a causal link $\beta \rightarrow \alpha$.

  The transformation resolving the unsupported assertion α is a 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 any previous commitment that has been made regarding the source of the causal support of α. For this reason, an assertion β is only considered as possible supporter if the action that added

β is a descendant of all tasks $\tau \in DT_\alpha$. Put otherwise, the supporter must originate from the refinement trees that were previously selected.

- **Delayed support from existing task.** Another possible source of supporting assertions is in the unrefined tasks of ϕ. 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., $\langle sv{=}v \rangle \in E^+_\tau$, and
  - there is enough time for the possible effect to occur before α, meaning that the addition of the temporal constraint $start(\alpha) - start(\tau) \geq \Delta_{PosEff}(\tau, \langle sv{=}v \rangle)$ does not 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 flaw, it makes a commitment to a subset of resolvers in the possible effects of τ. The choice of a supporting assertion for α will be delayed until τ is refined. 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 chronicle itself but constrains the planner to 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 α. For this reason, we only consider as a resolver a task $\tau \in TS_\alpha$ if, for any task $\tau' \in DT_\alpha$, τ is a descendant of τ'.

- **Delayed support from new 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 must appear by means of task refinement only. When $DT_\alpha = \varnothing$ 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 brought by inserting an instance a of A in ϕ and adding $task(a)$ 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 which assertion will support α is delayed.

To summarize, every assertion needs a causal supporter with and explicit 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 refining pending tasks in the plan, or requires the introduction of new actions outside of existing decomposition trees. In the two latter choices, the selection of the actual causal supporter is delayed until the task network is refined.

3.2.2 Unrefined Tasks

A task is *unrefined* if *(i)* it has been required either in the problem specification or as a subtask of an action in the plan, and *(ii)* it has not been refined by a *task refinement* transformation (Section 2.4.1). Given a chronicle $\phi = \langle \pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi \rangle$, any task τ appearing in π_ϕ is an unrefined task. Resolving an unrefined task requires selecting an action that achieves it and applying the corresponding task decomposition transformation. Our procedure extends the usual HTN task decomposition to account for the commitments previously made while resolving unsupported assertion flaws.

Let $\tau : \texttt{tsk}(x_1, \ldots, x_n)$ be an unrefined task, and PR_τ be the set of action templates A such that $task(A)$ is unifiable with τ. 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. In the resulting chronicle ϕ', the unrefined task τ has been replaced by a new action instance a.

Furthermore, we need to account for the possible commitments 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.

3.2.3 Possibly Conflicting Assertions

This flaw occurs when a chronicle $\phi = (\pi_\phi, \mathcal{F}_\phi, \mathcal{C}_\phi)$ has an instantiation that makes two assertions conflicting. We start by studying the case of persistence assertions and then consider change assertions.

Two persistence assertions $p_1 = \langle [t_1^s, t_1^e]\, sv_1 = v_1 \rangle$ and $p_2 = \langle [t_2^s, t_2^e]\, sv_2 = v_2 \rangle$ are conflicting when they concurrently require distinct values for the same state variable. More precisely they are *possibly conflicting* when the conjunction of these three conditions holds:

- they can refer to the same state variable, that is sv_1 and sv_2 are respectively of the form $sv(x_1, \ldots, x_n)$ and $sv(y_1, \ldots, y_n)$ and $\mathcal{C}_\phi \cup \{ x_1 = y_1 \wedge \cdots \wedge x_n = y_n \}$ is a consistent set of constraints,
- they can temporally overlap, i.e., $[t_1^s, t_1^e] \cap [t_2^s, t_2^e] \neq \varnothing$ is consistent with the temporal constraints in $\mathcal{C}_\phi$, and
- 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 cannot be conflicting in any 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 the following:

- 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 $y_{1..n}$ are the arguments of the state variables of p_1 and p_2 respectively,
- a temporal separation constraint, i.e., c is either $t_1^e < t_2^s$ or $t_2^e < t_1^s$, or
- a unification constraint $(v_1 = v_2)$ that requires the two values to be the same .

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 $[t_s]\, sv = v_s$,
- an intermediate persistence $\langle]t_s, t_e[\ sv = \textsc{undefined} \rangle$, where $\textsc{undefined}$ is a special value that is not unifiable with any variable including $\textsc{undefined}$ itself (i.e., $\forall v, \textsc{undefined} \neq v$), and
- an end persistence $[t_e]\, sv = v_e$.

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

For the purpose of identifying conflicts, an assignment assertion $\langle [t]\, sv := v \rangle$ is seen as a change assertion $\langle [t-1, t]\, sv : any \mapsto v \rangle$ where any is an unbounded variable that can take any value.

Conflicting assertions extend the threats of plan space planning with assertions spanning over temporal intervals. The resolvers are separation constraints on temporal as well as on object

variables in the two conflicting assertions.

3.3 Constraint Networks

FAPE makes an important use of constraint networks to manage the temporal aspects of the problem and to maintain partially instantiated chronicles. The constraint networks have to efficiently store the constraints accumulated by the planner and answer various queries necessary for the identification of flaws and finding resolvers.

For instance, finding whether assertions are possibly conflicting requires testing their possible temporal overlap and unification of their state variables. For this purpose we have two distinct constraint networks: one responsible for temporal reasoning and one for handling binding constraints. A special case of constraints are *duration constraints* that refer to variables present in both networks. such as the one enforcing travel time in Figure 1. We start by presenting the temporal and binding constraint networks independently, and then show how they are synchronized to handle duration constraints.

3.3.1 Temporal Constraint Network

Temporal variables (or timepoints) and temporal constraints are handled as a Simple Temporal Networks (STN) [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 at least d. The consistency and the minimal network of an STN can be computed in polynomial time.

The Floyd-Warshall All-Pairs Shorter Path algorithm [Flo62; War62] gives the minimal network where the minimum and maximum distances between all pairs of timepoints are explicitly computed. This is performed in $\Theta(|\mathcal{X}|^3)$. We use the incremental version of this algorithm, as presented by Planken [Pla08], that has a quadratic complexity of $O(|\mathcal{X}|^2)$.

We denote as $dist_{STN}(t_1, t_2)$ the minimal delay between t_1 and t_2, which is given in the minimal network. The maximum delay from t_1 to t_2 is simply $-dist_{STN}(t_2, t_1)$.

3.3.2 Binding Constraint 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 integer variables.
- $\mathcal{D}_{Bind}$ is a set of finite domains for each variable in $\mathcal{X}_{Bind}$. The initial domain of object variables is a subset of the objects in the problem having the type of the object variable. The initial domain of integer variables is the set of integers.
- $\mathcal{C}_{Bind}$ 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 constraint, 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 table that gives the allowed tuple of values for the variables $x_{1..n}$,
  - an inequality on a integer variable, e.g., $x_1 \leq 12$ where x_1 is a integer variable, or
  - 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. These constraints correspond to NP-complete CSPs. (Networks with inequality constraints alone are NP-Complete.)

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 $\mathcal{Q}_{Bind}$. Until $\mathcal{Q}_{Bind}$ is empty, propagation iterates overs the following three steps:

(i) A constraint c is extracted from $\mathcal{Q}_{Bind}$.

(ii) For any variable x involved in c, remove from $dom(x)$ any value that cannot satisfy this constraint.

For a relation constraint $c = \langle R(x_1, \ldots, x_{n-1}) = x_n \rangle$, we first remove from the relation any tuple of values $(v_1, \ldots, v_n)$ that is not achievable (i.e., $\exists i \in [1, n] \mid v_i \notin dom(x_i)$). 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 $\bigcup_{i \in [i,n]} dom(x_i)$.

(iii) If the domain of any variable x was updated during propagation, all constraints involving x are added to $\mathcal{Q}_{Bind}$.

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

The network is 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.
- *Separable* if they are not unified.
- *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, i.e., they have non intersecting domains or there is an inequality constraint between them.

3.3.3 Duration Constraints

Consider the constraint $t_2 - t_1 \geq \delta$ where t_1 and t_2 are timepoints and δ is an integer variable in the binding constraint network. This *duration constraint* involves variables of both networks. For instance, the introduction of an instance a of the move action of Figure 1, would result in the

following constraints:

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

where r, d and d' are object variables, $\textit{start}(a)$ and $\textit{end}(a)$ 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.

A duration constraint is handled as follows:

- When a new constraint $d \geq t_2 - t_1$ is added or inferred in the temporal network, we remove from the domain of δ any value that would force a duration greater than d:

  $$\textit{dom}(\delta) \leftarrow \{\, v \mid v \in \textit{dom}(\delta) \ \wedge\ d \geq v \,\}$$

  If the domain of δ is modified by this operation, it will trigger propagation in the binding network.
- When the domain of δ is modified in the binding constraint network, a new temporal constraint $t_2 - t_1 \geq \min\{\, v \mid v \in \textit{dom}(\delta) \,\}$ is added to the STN. I.e the least constraining instantiation of δ is propagated into the temporal network.

To handle the more general form of a duration constraint $t_2 - t_1 \geq \textit{fn}(x_1, \ldots, x_n)$ where t_1 and t_2 are timepoints of the temporal network, $x_1, \ldots, x_n$ are object variables in the binding network, and $\textit{fn}$ is a function given as or transformed into a table, we reify $\textit{fn}(x_1, \ldots, x_n)$ by *(i)* introducing a new integer variable δ, *(ii)* imposing the relation constraint $\textit{fn}(x_1, \ldots, x_n) = \delta$ in the binding network and *(iii)* enforcing $t_2 - t_1 \geq \delta$ as above.

Hence the constraint $t_{end} - t_{start} = \textit{travel-time}(r, d, d')$ of Figure 1 is effectively reformulated into the following conjuncts that can be handled separately:

$$\textit{travel-time}(r, d, d') = \delta \quad \wedge \quad t_{end} - t_{start} \geq \delta \quad \wedge \quad t_{start} - t_{end} \geq -\delta$$

3.4 Search Space: Properties and Exploration

Here we analyze the characteristics of the search space of Fape.

Search space dimension. The search space of plan-space planners is infinite, since they explore the set of plans which is infinite [GNT04, Chap. 5]. Indeed, even for a simple 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 the destination. Similarly, the presence of recursive methods can make the search space of HTN planners infinite [EHN94]

Since our search procedure is an extension to plan-space planning and can represent HTN problems, the search-space of Fape is infinite as well. Under 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 iterative deepening meet those expectations. More interestingly, best-first search (and A* in particular) also meet those expectations as long as the addition of an action to the chronicle has a strictly positive cost for its evaluation function. Indeed when the number of actions in a plan approaches 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 4.4.1).

Search space acyclicity. Another condition for the completeness of the search is either an acyclic space or an adequate handling of cycles, if any, to avoid infinite loops.

Proposition 3.1. *The search space of* FAPE *is acyclic.*

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

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 results in one flaw being solved (even in the case where the resolution of an unsupported assertion is delayed, this unsupported assertion is only reconsidered when at least one other flaw has been fixed). Since there are only a finite number of flaws in a chronicle, 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 consider the soundness and completeness of our planning procedure.

Proposition 3.2. FAPE *is sound and complete.*

Proof (Sketch). The full version of this proof is given in Appendix A.1.

For soundness, we remark that each type of flaw identifies one of the conditions of Definition 2.4. Thus if a plan has no flaw, it fulfills all three requirements. The requirement that the solution plan be *reachable* from the initial chronicle is always fulfilled as all resolvers are defined in terms of the allowed transformation.

Given that the search space is acyclic and that the exploration is methodical, showing completeness requires us to show that a solution appears in the search space if one exists. We show this by proving that, for each flaw, the set of resolvers is complete and does not overlook any solution plan. □

Discussion. The definition of resolvers results in a top down approach to hierarchical planning, similar to that of other HTN planners. Indeed, the search starts from an existing task network, which, at the end, is entirely decomposed over the course of planning. In practice, this means that any action in the plan is inserted after the action it possibly descends from. This scheme has the advantage of being well documented and understood.

This decomposition procedure is integrated into generative planning, using partially instantiated plans. As a result, the planner is able to handle temporal planning problems spanning from fully generative to fully hierarchical, together with mixed specification of generative and hierarchical problems.

4 Search Control

Section 3.4 defined the search space of FAPE, obtained by extending the PSP algorithm to handle hierarchical decomposition. Compared to most planners, the search space of constraint-based planners (including FAPE) is much more compact as the lifted representation allows representation of many partial plans 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 good search strategies to perform efficiently. In this section, we detail the strategies used for shaping and exploring the search space in FAPE. These can be roughly separated into 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 a very different search space than the one resulting from resolving conflicting assertions first.
- the strategy of *which partial plan to consider next* is critical to guide exploration of the search space towards a 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 would be 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 and restricting the search space that will be explored given the partial plan selection strategy. We first explain the techniques used for automatic inference in FAPE (Sections 4.1 to 4.3). More specifically, Section 4.2 will detail a reachability analysis to infer what cannot be achieved from a partial plan due to its temporal and hierarchical features and Section 4.3 will provide techniques to reason about the current causal structure of a plan. Finally, Section 4.4 will focus on the strategies for flaw and plan selection.

4.1 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 assertion can be refined into many ground ones. This can be detrimental as most heuristic computations for planning rely on 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 instantiations 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 $id = \langle \mathtt{act}(\mathtt{c1}, \ldots, \mathtt{cn}) \rangle$, 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 1 gives an example of this relation for the move action of Figure 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 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 $\text{move}(\texttt{r1}, \texttt{d1}, \texttt{d2})$.

A (lifted) action $a : \text{act}(x_1, \ldots, x_n)$ 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 every ground action that a might become once all its parameters are instantiated:

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

To enforce, this relationship, every time a (lifted) action instance $a = \langle \text{act}(x_1, \ldots, x_n) \rangle$ 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.

This instantiation variable has two main benefits. First it allows us 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 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 $\tau : \texttt{tsk}(y_1, \ldots, y_n)$ is associated with a refinement variable R_τ with values being 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 $id = \text{act}(\texttt{c1}, \ldots, \texttt{cn})$ 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 and 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 a grounded 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.

4.2 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. This further 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 incrementally expanded 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 interdependent actions. Indeed, if two actions A and B are interdependent (as in Figure 4), inferring that B is reachable requires knowing that A is reachable, which would require prior knowledge that B is reachable. Planners such as POPF get around this problem by further relaxing delete-free problems: each durative action A is split in two instantaneous at-start and at-end actions, A_{start} and A_{end}. A_{start} contains only the start conditions and start effects of A, while A_{end} contains only the end conditions and end effects of A. Since A_{start} does not contain the end conditions A this eliminates any possibility of inter-dependency [Col+08; CMR13]. To illustrate how reachability works, in the above example if A_{start} were reachable, B_{start} would be reachable, allowing B_{end} to be reachable. This, in turn, allows A_{end} to be reachable. This works fine as long as B is shorter than A. However, it is overly optimistic when B is longer than A, leading the analysis to conclude that both actions are reachable when they are not.

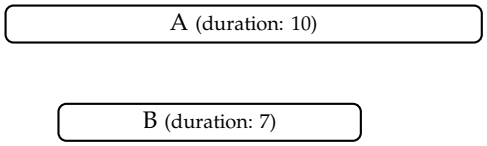


Figure 4: Two interdependent actions: A with a start effect x and an end condition y, and B with a start condition x and an end effect y.

Hierarchical problems introduce many of these kinds of problems, because a higher level task requires that the subtasks be contained within it, and the subtasks require the presence of the higher level task. An illustration of such interdependencies is given in Figure 5.

In this section, we describe a reachability analysis technique that takes into account the hierarchical properties of a problem and supports interdependent actions with no additional relaxation. As in most existing planners, we consider delete-free actions in our relaxed model. To give an intuition of what a delete-free relaxation would be in our planning model, 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

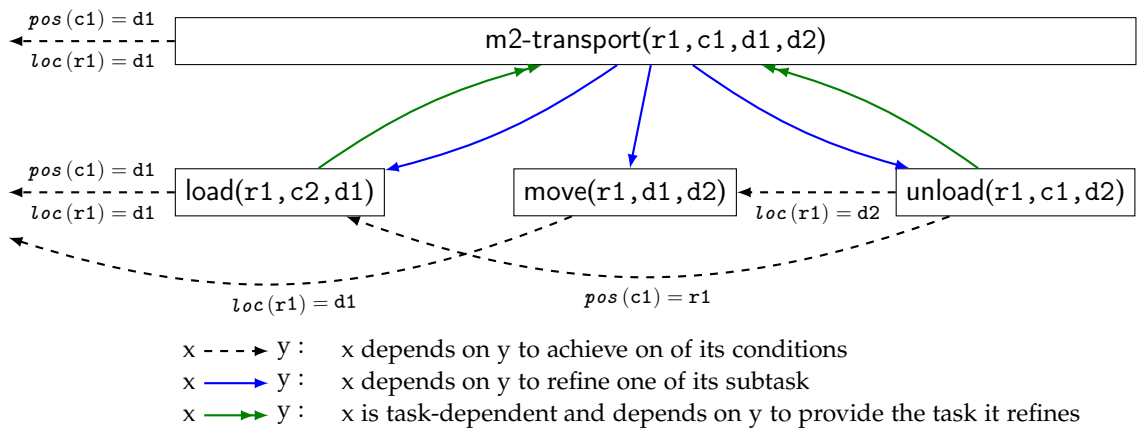


Figure 5: Dependencies between actions in a plan achieving a single `transport(c1,d2)` task. The m2-transport action is the one of Figure 2. Its three subtasks are refined by a move action (Figure 1) and two *task-dependent* load and unload actions.

$\langle sv{=}\mathtt{a}\rangle$ 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 partial plans as well as to detect dead ends in the search.

4.2.1 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 refining 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 fluents 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 reachable subsets of F and A that can be reached from the facts in I.

$\mathsf{move}_{flat}(\mathtt{r1},\mathtt{d1},\mathtt{d2})$

conditions:	$[t_{start}]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d1}$
	$[t_{start}]\ \mathit{occupant}(\mathtt{d1}) = \mathtt{r1}$
	$[t']\ \mathit{occupant}(\mathtt{d2}) = \mathit{nil}$
effects:	$[t_{end}]\ \mathit{loc}(\mathtt{r1}) = \mathtt{d2}$
	$[t]\ \mathit{occupant}(\mathtt{d1}) = \mathit{nil}$
	$[t_{end}]\ \mathit{occupant}(\mathtt{d2}) = \mathit{nil}$
	$[t_{start}]\ \mathit{started}(\mathtt{move}(\mathtt{r1},\mathtt{d1},\mathtt{d2})) = \top$
	$[t_{end}]\ \mathit{ended}(\mathtt{move}(\mathtt{r1},\mathtt{d1},\mathtt{d2})) = \top$
constraints:	$\mathit{connected}(\mathtt{d1},\mathtt{d2})$
	$t_{end} - t_{start} = 10$
	$t_{start} < t < t' < t_{end}$

Figure 6: Flattened version of the move(r1,d1,d2) operator whose template is given in Figure 1. The implicit temporal constraints are shown explicitly here.

Elementary actions. We start by transforming our actions (both high-level and primitive) into a set of simpler actions with only single effects. Our objective is to obtain delete-free actions that still encompass temporal and hierarchical aspects of the original action. We start by associating any action a of the planning domain with a flat action a_{flat} with a set of conditions and effects 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 assignment 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 $\langle [t_{start}]\, \mathit{started}(\tau_a) = \top\rangle$ and the effect $\langle [t_{end}]\, \mathit{ended}(\tau_a) = \top\rangle$
- if a is task-dependent and its task is τ_a, then a_{flat} has the additional condition $\langle [t_{start}]\, \mathit{required}(\tau_a) = \top\rangle$
- for every subtask $\langle [t,t']\, \tau\rangle$ of a, a_{flat} has:
  - two additional conditions $\langle [t]\, \mathit{started}(\tau) = \top\rangle$ and $\langle [t']\, \mathit{ended}(\tau) = \top\rangle$
  - one additional effect $\langle [t]\, \mathit{required}(\tau) = \top\rangle$
- 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 $\langle [t_s,t_e]\ \tau\rangle$ yields two conditions $[t_s]\, \mathit{started}(\tau) = \top$ and $[t_e]\, \mathit{ended}(\tau) = \top$. 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 is however not a problem as this transformation is simply meant to expose some hierarchical features of the problem for an optimistic reachability analysis.

The flat actions are still temporally complex and might feature numerous timepoints related by temporal constraints. As a second preprocessing step, each flat action is split into simpler

$\mathsf{move}_{loc(\mathtt{r1})}(\mathtt{r1},\mathtt{d1},\mathtt{d2})$

conditions: $[-9]\ loc(\mathtt{r1}) = \mathtt{d1}$
$[-9]\ occupant(\mathtt{d1}) = \mathtt{r1}$
$[0]\ occupant(\mathtt{d2}) = nil$
$[-9]\ required(\mathtt{move}(\mathtt{r1},\mathtt{d1},\mathtt{d2})) = \top$

effects: $[1]\ loc(\mathtt{r1}) = \mathtt{d2}$

constraints: $connected(\mathtt{d1},\mathtt{d2})$

$\mathsf{move}_{occupant(\mathtt{d1})}(\mathtt{r1},\mathtt{d1},\mathtt{d2})$

conditions: $[0]\ loc(\mathtt{r1}) = \mathtt{d1}$
$[0]\ occupant(\mathtt{d1}) = \mathtt{r1}$
$[9]\ occupant(\mathtt{d2}) = nil$
$[0]\ required(\mathtt{move}(\mathtt{r1},\mathtt{d1},\mathtt{d2})) = \top$

effects: $[1] occupant(\mathtt{d1}) = nil$

constraints: $connected(\mathtt{d1},\mathtt{d2})$

Figure 7: The first two elementary actions generated from $\mathsf{move}_{flat}(\mathtt{r1},\mathtt{d1},\mathtt{d2})$

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:

- 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, which 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 a least-constraining timing constraint on when c is needed relative to the effect e. By least-constraining, we mean the latest time at which c can be required to be true given the temporal constraints. For a condition $\langle [t_c] sv' = v' \rangle$ in a_{flat}, this is achieved by adding to a_e a condition $\langle [1 + max(t_c - t_e)] sv' = v' \rangle$ where $max(t_c - t_e)$ is the maximal delay from t_e to t_c as given by the temporal constraints in the action.[4]

Figure 7 shows two elementary actions generated for the flat move of Figure 6. 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. It is meant to capture the fluents that are known to hold at a given time, regardless of whether they derive from timed initial literals or actions in the current partial plan. A 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 $dist_{STN}(\mathcal{O}, t_2)$.

[4]Recall that in a delete-free problem, if a fluent holds at a given time t, it will hold for any subsequent time $t' \geq t$.

- 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 of 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) 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.

4.2.2 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 when or after an effect of the action is achieved. The end condition y of the action A of Figure 4 is an *after-condition*. Because the effect of our elementary actions are placed at time 1, 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 means that the after-conditions of at-start actions are ignored. As a result, a start effect can be considered as reachable even when an end condition is not. In Figure 4, the action A would indeed become an 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 next subsection describe a technique for reachability analysis that does not need any additional relaxation, thus taking all after-conditions into account.

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* nodes (i.e. actions or fluents) from which to start propagation (lines 4-12). The obvious candidates are fluents known to be true at

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

```
A ← Elementary actions
F ← Fluents
I ← Initial timed fluents
Q ← ∅                    ▷ Priority queue of (action/fluent, time) ordered by increasing time
for all f ∈ F do
    reachable(f) ← ⊥
for all ⟨[t] f⟩ ∈ I do
    Q ← Q ∪ {(f, t)}
for all a ∈ A do
    reachable(a) ← ⊥
    if a has no before-conditions then
        Q ← Q ∪ {(n, 0)}

while Q non empty do
    DijkstraPass
    for all a ∈ A do
        for all ⟨[δ] f⟩ ∈ after-conditions of a do
            if ¬reachable(f) then
                (A, F) ← RecursivelyRemove(a)
            else if ea(a) < ea(f) − δ then
                Q ← Q ∪ { (a, ea(f) − δ) }
    for all a ∈ A do
        if a is late then
            (A, F) ← RecursivelyRemove(a)

procedure DijkstraPass
    while Q non empty do
        (n, t) ← pop(Q)
        if n already expanded in this pass then
            continue
        if reachable(n) ∧ ea(n) ≥ t then
            continue
        reachable(n) ← ⊤
        ea(n) ← t
        if n is an action with the effect ⟨[1] f⟩ then
            Q ← Q ∪ { (f, t + 1) }
        else
            for all a ∈ A with a condition on the fluent n do
                if all before-conditions of a are reachable then
                    t′ ← max_{⟨[δ] f⟩ ∈ cond(a)} ea(f) − δ
                    Q ← Q ∪ {(a, t′)}
```

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 in 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 conditions are all 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 $\langle [1]\, f \rangle$, the pair $\langle f, t+1 \rangle$ 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 onto 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 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, the 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.

4.2.3 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.

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 4.1 (Causal loop)**.** We denote as a causal loop a cycle of actions and fluents $f_0 \rightarrow A_0 \rightarrow f_1 \rightarrow A_1 \dots A_n \rightarrow f_0$, 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 that is respectively the delay from when an action A_i can 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.

We say that a causal loop is self-supporting if its length (i.e. the sum of the delays on its edges) is less than or equal to 0.

A causal loop is said to be unfeasible if its length is strictly positive.

The notion of causal loop is crucial in the understanding of problems with interdependent actions. The actions of Figure 4 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.

On the other hand, if B had a duration of 12, 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 4.1. *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.2. 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 4.2. *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.3. The intuition is that the gap between non-late and late nodes appeared because late nodes are delaying each others 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 initial timed literals or negative cycle) would have resulted in n being less than d_{max} away from a non-late node. □

It follows from propositions 4.1 and 4.2 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). As we will see in the experiments (Section 5.1.1), this worst case doesn't tend to occur in practice; typically, quiescence occurs after a relatively small number of iterations. The cost 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 the algorithm 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 temporal planning graph of POPF: *(i)* the separation of durative actions into at-start and at-end instantaneous actions is done by the transformation into elementary actions; *(ii)* 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 *(iii)* any end condition appearing in the elementary action of a start effect would be ignored because it would be an after-condition. This model, that we call R^+, is thus a direct adaptation of the techniques used in POPF to our richer action representation.

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 temporal planning graph, with no additional computational overhead.

4.2.4 Exploiting 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,
- $ea_\phi : (Ra_\phi \cup Rf_\phi) \rightarrow \mathcal{N}$ 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.

These 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 worst case complexity of the incremental version is unchanged, our implementation suggests that it avoids a lot of redundant computation. 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:

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

  where $\mathcal{O}$ is the temporal origin and $dom(cond(\alpha))$ denotes all possible instantiations of the fluent that is required by α.
- We check that all unrefined tasks can be refined by a reachable action. This is done by restricting the domain of any refinement variable (Section 4.1) 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 task flaws, we disregard any resolver involving an action with no reachable instances. For instance, if there is no instances 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 (to be introduced in Section 4.3) 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 4.1, the domain of these variables is constrained to be a subset of Ra_ϕ. This indirectly constrains the parameters of any newly added action to respect reachability requirements.

4.3 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 $x \rightarrow y$ 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 (i.e. an incoming causal link),
- 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 4.2 (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}.

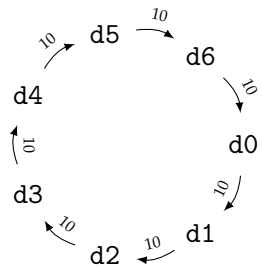


Figure 8: Example DTG of $loc(r)$: where the location in which the robot r can navigate are organized in a circular pattern. Moving from one place to the next takes 10 time units.

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. Over its temporal interval, a causal chain fully constraints the evolution of its state variable.

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 instantiation overlaps. 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 4.3 (DTG)**.** The Domain Transition Graph (DTG) of a state variable sv is a directed graph (V, T) where V is composed of the values that can be taken by sv and a special node ANY. T is a set of allowed transitions from one value to the other. An edge in T 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).

DTGs are used to reason on the changes that can be made to a state variable sv through the addition of new actions in a partial plan. 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 of sv contains an edge $v_1 \xrightarrow{min(t_2 - t_1)} v_2$,
- if the action contains an a priori supported assertion $\langle [t]\, sv{:=}v \rangle$, then the DTG of sv 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 8.

4.3.1 Possible 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

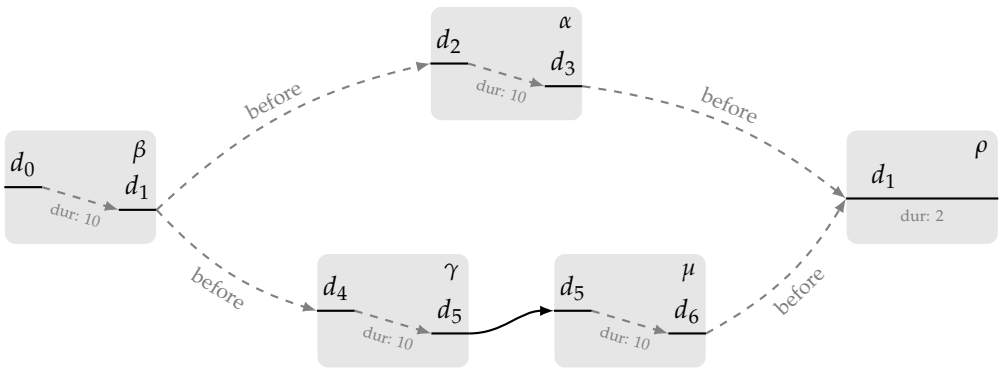


Figure 9: Partial view of a causal network of a state variable $loc(r)$ with 4 change assertions $(\beta, \alpha, \gamma, \mu)$ and one persistence condition ρ, all on the same state variable $loc(r)$. There is a causal link from γ to μ, β is temporally constrained to be before α and γ, and ρ is temporally constrained to be after α and μ. We further suppose β to be initially supported. This causal network is to be considered together with the DTG of Figure 8.

links going through statements not yet in the plan. We refer to the candidates for such supporting assertions as *possible supporters*.

Definition 4.4 (Possible supporter)**.** Given the causal network G_ϕ of a partial plan ϕ, a change assertion β is a *possible supporter* of an unsupported assertion α if there is a set of statements $\{s_1, \ldots, s_n\}$ and a chain of causal links $\beta \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \alpha$ that can be added to G_ϕ.

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

- the state variables of α and β are unifiable.
- there is a feasible transition in the DTG from the value produced by β to the value required by α.
- adding a chain of causal links 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 on the same state variable.

In the causal network example of Figure 9, the possible supporters of α would be β and μ because β/μ can come before α and the DTG has a path from d_1/d_6 to d_2. The possible supporters of γ would be β and α because β/α can come before γ and the DTG has a path from d_1/d_3 to d_4. The possible supporters of ρ are α and μ; indeed any causal chain from β to ρ would be threatened by α and by the causal chain of γ and μ.

In search, we restrict resolvers of an unsupported condition α to the assertions in $\mathcal{S}_\alpha$. This removes infeasible resolvers and thus reduces the number of branches in the search tree.

4.3.2 Deriving Constraints from Potential 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 $\beta \rightarrow \cdots \rightarrow \alpha$. The length of the causal chain depends on the change assertions needed to go from the value produced by β (denoted $\mathit{eff}(\beta)$) to the value needed by α (denoted $\mathit{cond}(\alpha)$). Therefore, $\mathit{start}(\alpha)$ must be after $\mathit{end}(\beta)$ with a delay depending on the times of $\mathit{eff}(\beta)$ and $\mathit{cond}(\alpha)$. More formally, this requirement is expressed by the following inequality,

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

where $\mathit{dist}_{STN}(\mathcal{O}, t)$ is the minimal delay in the STN between the origin of time $\mathcal{O}$ and the time point t and $\mathit{dist}_{DTG}(x, y)$ 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 $\mathit{start}(\alpha)$ to be greater or equal to the right side of the inequality.

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

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

Again this inequality is enforced by adding in the STN a minimal delay constraint between $\mathit{end}(\beta)$ and $\mathit{start}(\alpha)$.

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

Example 4.1. Let us now consider what applying those rules on the causal network of Figure 9 would allow us to infer. Assuming that $\mathit{dist}_{STN}(\mathcal{O}, st_\beta) = 0$, we would infer the following temporal constraints:

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

The important catch is the detection that the persistence condition $\rho = \langle loc(r) = d_1 \rangle$ cannot be satisfied before time 80. This is because 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. □

4.3.3 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 in order to support an unsupported assertion.

For each unsupported assertion 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 10 gives an example of the minimal causal chain needed to support the persistence condition ρ from Figure 9. Building such a causal chain requires the addition of 4 change assertions, resulting in as many new open goals to be solved.

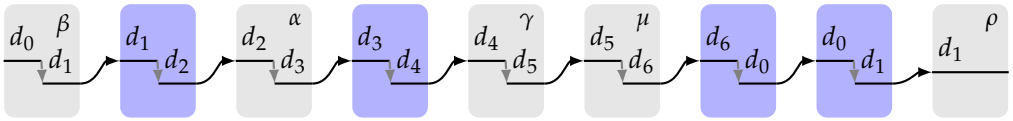


Figure 10: One possible causal chain to support the persistence condition $\rho = \langle loc(r) = d1 \rangle$. In blue are temporal assertions that would need to be added for the causal chain to be complete.

We now describe how we compute the heuristic value, $hc(\alpha)$, 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:

$$hc(\alpha) = \min_{\langle sv{=}v \rangle \in dom(cond(\alpha))} hc(\langle sv{=}v \rangle, \alpha) \tag{1}$$

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.

We define $hc(f, \alpha)$, the cost of achieving the ground condition $f = \langle sv{=}v \rangle$ of an assertion α. The cost of building a causal chain is expressed recursively as the cost of having its rightmost link plus the cost of building a chain up to this rightmost link.

$$hc(\langle sv{=}v \rangle, \alpha) = \begin{cases} 0 & \text{if } \alpha \text{ is a priori supported} \\ \min\limits_{\langle sv{=}v' \mapsto v) \rangle \in dom(\beta)} hc(\langle sv{=}v' \rangle, \beta) & \text{if } \exists \text{ a causal link } \beta \to \alpha \\ \min \begin{cases} \min\limits_{\gamma \in \mathcal{S}_\alpha,\ \langle sv{=}v' \mapsto v \rangle \in dom(\gamma)} hc(\langle sv{=}v' \rangle, \gamma) \\ \min\limits_{e = \langle v' \to v \rangle \in DTG(sv)} c(e) + hc(\langle sv{=}v' \rangle, \alpha) \end{cases} & \text{otherwise} \end{cases} \tag{2}$$

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 9, 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 $\gamma \rightarrow \alpha$. In this case, the cost is that of achieving γ with such an instantiation. Second, there might be a possible action producing $f = \langle sv{=}v \rangle$ by adding a change assertion $[t, t']\, sv : v' \mapsto v$. Such a change would appear as an edge e in the DTG of sv. Since 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 1 this cost would be 3 since its insertion would result in 3 new assertions. Furthermore, we still need to build the causal chain to achieve the value $\langle sv{=}v' \rangle$ for α.

Example 4.2. Considering the causal network of Figure 9, 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 $\mathtt{move}(r, \mathtt{d1}, \mathtt{d2})$ 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 $\mathtt{move}(r, \mathtt{d0}, \mathtt{d1})$ 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 assertions into the partial plan.

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

In practice 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 9 is given in Figure 11. Edges in the graph represent the different possible transitions defined in equation (2). We distinguish a causal link from an existing assertion (green with a cost of 0) from a causal link from an additional assertion (dashed 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 (full circles). Finding a minimal causal chain to an unsupported assertion α is equivalent to finding the shortest path in the graph from any *a priori* supported node to any node representing an instantiation of α.

In the example of Figure 11, we can easily find the causal chain of Figure 10 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 $hc(\rho) = 12$. This corresponds to the causal chains shown in Figure 10.

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 progresses 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. $hc(\alpha)$).

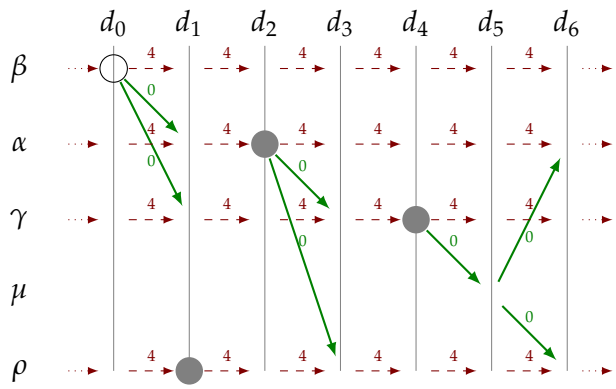


Figure 11: Virtual graph used for computing the minimal causal chain of the assertions of Figure 9.

4.4 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 objectives. 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.

As 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 tree. A search strategy is composed of two schemes that dictate the choices made in those two cases.

As can be expected, a good strategy is not universal as it must take into account many specifics of the problem at hand. Because we support such a wide range of planning problems, from generative to HTN problems, we define two strategies. The first one aims at being very general and having good performance on a wide range of problems while the second is specifically tailored for fully hierarchical problems.

4.4.1 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 *fn*. 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 *fn* 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 $\min_{\phi \in Q_{All}} fn(\phi) < (1+\epsilon) \times \min_{\phi \in Q_{Chi}} fn(\phi)$ then the next partial plan is the one with the lowest *fn* value in Q_{All}. Otherwise it is the one with the lowest *fn* value in Q_{Chi}. In practice, it means the

search is restricted to the children of the last expanded node 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 for the inaccuracies in the definition of *fn*.

The priority function $fn(\phi)$ 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 be seen as a normalized version of the number of actions in the plan as other parts of *fn* depend on the number of 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 *fn*. 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 assertions needed to reach the minimal causal chain, as defined in Section 4.3.3.

This definition of the priority function *fn* 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 following 4. 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 provides a heuristic estimation of the search effort required.

Flaw Selection. The flaw selection strategy aims at sorting flaws in order to select the one whose resolution would the more 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 the 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 rule favors 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 task refinement, 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 *fn* function is more 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 fewest resolvers.

Furthermore, it should be noted that if there is a flaw with no resolver, then this partial plan is necessarily a dead-end and the planner can proceed with the next partial plan.

4.4.2 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 refinements in the order defined in the domain and commit to them until they are proved unsound. The planner also follow an early commitment strategy through its flaw selection strategy, which is meant to detect early any inconsistencies in the current plan.

Plan selection. Plan selection is done 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 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. The rest of the search decisions, such as the choice of how to resolve a threat, is left 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 choose 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 give priority to the maximally abstract ones.
6. Prefer flaws with the fewest 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 later actions of the plan early during search by solving a flaw with a single resolver.

5 Empirical Evaluation

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

In this section, we first study how each of the techniques we have described contribute to the overall efficiency of the system. We then compare FAPE with state-of-the-art temporal planners from the International Planning Competition (IPC). We show FAPE to be competitive in a fully generative and domain-independent setting and that the addition of hierarchical knowledge further improves its performance.

5.1 Evaluation of the Different Components of the Planner

5.1.1 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 *(i)* it generalizes the delete-free analysis done by classical planners to a temporal setting and *(ii)* 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_k, with $k = 5$ and $k = 1$, denotes the configuration where the number of iterations is limited to k. 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 more optimistic 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.

To allow for an unbiased comparison, the reasoning on causal networks (Section 4.3) is deactivated in all configurations as it requires reasoning on the ground problem (which is inaccessible for the last configuration).

Test Domains. 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*, *blocks*, *driverlog* and *hiking*

domains are the eponymous domains from the International Planning Competition and are all temporally simple (i.e. they have no after-conditions). The domain definitions of those problems have been manually translated into ANML and their problems were automatically translated by domain dependent parsers.

The *handover* domain is a robotics problem presented by Dvorák et al. [Dvo+14], the *dock-worker* domain is the dock worker domain of Ghallab, Nau, and Traverso [GNT04]. *race* is a robotics domain adapted from CHIMP [Sto+15], where a waiter-robot must serve clients. It notably features navigation constraints expressed using hierarchical features and deadlines specifying when 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 requires 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 robot if its hands are full. *baking* is a domain presented by Cushing et al. [Cus+07b] of baking pottery in kilns where the action of baking must be concurrent with an action showing that the kiln is switched on.

Hierarchical versions of the domains have their names appended with '-FullHier' or '-PartHier' if they are fully or partly hierarchical respectively.

Results and Discussion. Table 2 and Figure 12 present the number of problems solved using different reachability models. On generative problems (first half of the table), only a few domains benefit from reachability analysis: *handover*, *logistics* and *satellite-tw*. On other generative domains, the cost of grounding the problem is slightly higher or on-par with the benefits of using reachability analysis. As expected, all configurations using reachability analysis perform identically on temporally simple problems.

This state of affairs changes on hierarchical problems as they contain many interdependencies. In those domains, 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 $\varnothing$).

Table 3 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 complete iteration is often sufficient to capture most of the problematic after-conditions. However, on more complex problems such as *dock-worker*, *hiking*-FullHier, *handover*-FullHier, more iterations are beneficial either in terms of detected unreachable actions or in terms of solved problems. The initial propagation can take as much as 58 iterations for R_∞. The subsequent propagations are typically faster as they are made incrementally. As expected, a single iteration was sufficient to converge on all temporally simple problems.

5.1.2 Evaluation of other components

We next evaluate how each of the techniques presented 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 with all the techniques previously discussed enabled. It uses

	R_∞	R_5	R_1	R^+	$\varnothing$
(IPC4) airport-GEN	**5**	**5**	**5**	**5**	**5**
(IPC4) airport-tw-GEN	**6**	**6**	**6**	**6**	**6**
(IPC2) blocks-GEN	24	24	24	24	**26**
(LAAS) dock-worker-GEN	**9**	**9**	**9**	**9**	**9**
(IPC8) driverlog-GEN	**1**	**1**	**1**	**1**	**1**
(LAAS) handover-GEN	**4**	3	3	3	3
(IPC2) logistics-GEN	**7**	**7**	**7**	**7**	4
(IPC4) pipesworld-dl-GEN	5	5	5	5	**6**
(IPC5) rovers-GEN	34	34	34	34	**38**
(IPC4) satellite-tw-GEN	**9**	**9**	**9**	**9**	8
(IPC8) satellite-GEN	16	16	16	16	**17**
Total Generative	120	119	119	119	**123**
(IPC2) blocks-FULLHIER	**10**	**10**	9	4	9
(IPC2) blocks-PARTHIER	28	28	28	30	**32**
(LAAS) dock-worker-FULLHIER	**22**	**22**	**22**	20	20
(LAAS) dock-worker-PARTHIER	**12**	**12**	**12**	**12**	**12**
(LAAS) handover-FULLHIER	**10**	1	1	1	1
(IPC8) hiking-FULLHIER	**20**	13	12	12	12
(IPC2) logistics-PARTHIER	**28**	**28**	**28**	27	27
(LAAS) race-FULLHIER	**13**	**13**	**13**	**13**	**13**
(IPC8) satellite-PARTHIER	**18**	**18**	**18**	**18**	**18**
(LAAS) baking-PARTHIER	**6**	**6**	**6**	**6**	**6**
(IPC8) turnandopen-FULLHIER	**8**	**8**	**8**	0	0
Total Hierarchical	**175**	159	157	143	150
Total	**295**	278	276	262	273

Table 2: Number of solved problems for various domains with a 30 minutes timeout. The best results are shown in bold.

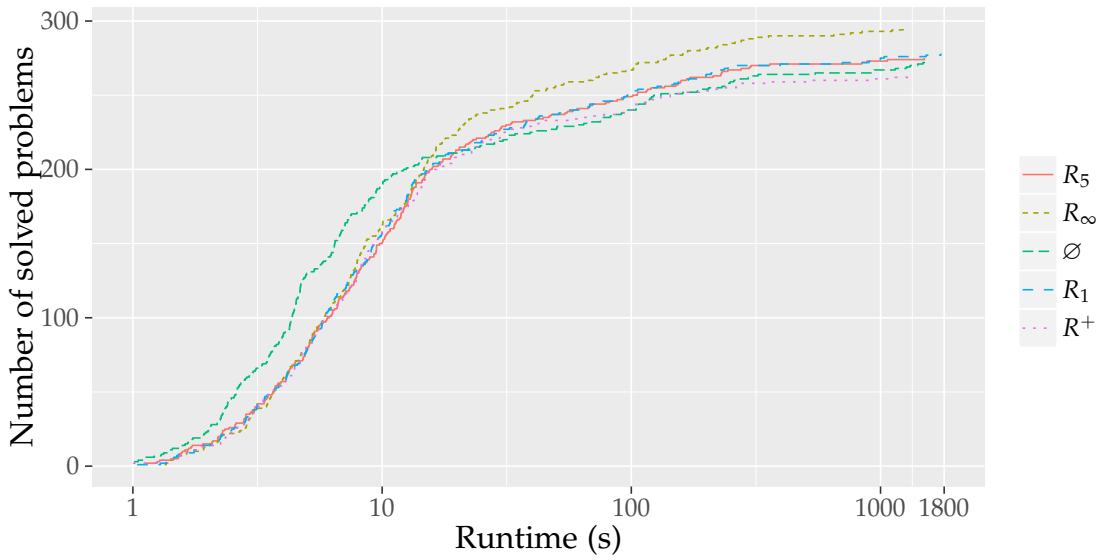


Figure 12: Number of solved tasks by each configuration within a given time amount.

the *general search strategy* (Section 4.4.1) by default and switches to the *forward hierarchical strategy* (Section 4.4.2) 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. causal networks and instantiation/refinement variables). This configuration is essential to measure the benefits and penalties for grounding the problem.
- $\mathcal{C}_{A^*}$ use A^* in place of our variant of A^ϵ (Section 4.4.1).
- $\mathcal{C}_{\neg DelCk}$ 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 an unsupported assertion. Hence it can result in additional resolvers being considered for supporting an unsupported assertion (Section 3.2.1).
- $\mathcal{C}_{\neg DecVars}$ does not use refinement variables to disregard the resolvers of unrefined tasks that involve unreachable actions (Section 4.1).
- $\mathcal{C}_{\neg CN}$ does not use the causal network to *(i)* infer constraints, *(ii)* prune impossible resolvers and *(iii)* improve the heuristic with needed assertions (Section 4.3).

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 the dominance of the current strategy as it has evolved with the rest of the system.

Overview of results. We start by giving a broad overview of the results given in Table 4. The table contains the number of problems solved by each configuration with a 30 minutes timeout

	R_∞	R_5	R_1	R^+	# iter R_∞
(IPC4) airport-GEN	**13.6**	**13.6**	**13.6**	**13.6**	1
(IPC4) airport-tw-GEN	**53.4**	**53.4**	**53.4**	**53.4**	1
(IPC2) blocks-GEN	0	0	0	0	1
(LAAS) dock-worker-GEN	**55.3**	**55.3**	**55.3**	0	6.0
(IPC8) driverlog-GEN	**41.8**	**41.8**	**41.8**	**41.8**	1
(LAAS) handover-GEN	**88.8**	87.0	87.0	55.6	29.0
(IPC2) logistics-GEN	**39.4**	**39.4**	**39.4**	**39.4**	1
(IPC4) pipesworld-dl-GEN	0	0	0	0	1
(IPC5) rovers-GEN	**45.7**	**45.7**	**45.7**	**45.7**	1
(IPC4) satellite-tw-GEN	**1.3**	**1.3**	**1.3**	**1.3**	1
(IPC8) satellite-GEN	0	0	0	0	1
(IPC2) blocks-FULLHIER	**83.1**	**83.1**	**83.1**	0	7.4
(IPC2) blocks-PARTHIER	**79.9**	**79.9**	**79.9**	0	2.0
(LAAS) dock-worker-FULLHIER	**80.8**	48.8	48.8	0	36.7
(LAAS) dock-worker-PARTHIER	**76.3**	58.8	58.8	0	57.7
(LAAS) handover-FULLHIER	**99.0**	87.4	87.4	0	47.2
(IPC8) hiking-FULLHIER	**74.1**	**74.1**	**74.1**	0	24.7
(IPC2) logistics-PARTHIER	**95.7**	**95.7**	**95.7**	10.4	2.8
(LAAS) race-FULLHIER	**94.3**	**94.3**	**94.3**	0	22.5
(IPC8) satellite-PARTHIER	**15.3**	**15.3**	**15.3**	**15.3**	2.0
(LAAS) baking-PARTHIER	**87.6**	**87.6**	**87.6**	0	5.8

Table 3: 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 to the original number of ground actions. Those values are then averaged over all instances of a domain. The last column gives the average number of iterations needed by R_∞ to converge on its initial propagation.

and a memory limit of 3GB.

The first important point is to see that while all features contribute to the efficiency of the planner, none is critical to its overall performance. Indeed, the absence of a feature resulted in 11 to 53 less problems being solved which is only a small subset of the overall 318 problems solved by the full configuration. However, the difficulty of a planning problem is typically exponential in the number of goals in the problem, so solving a few additional problems for a domain therefore has more significance than it might seem.

As expected the $\mathcal{C}_{gen}$, $\mathcal{C}_{\neg DelCk}$ and $\mathcal{C}_{\neg DecVars}$ variants behave exactly as $\mathcal{C}_{full}$ on generative problems. The use of our *forward hierarchical strategy* is however critical in some hierarchical domains, as it can be seen by the poor performance of $\mathcal{C}_{gen}$ on the *hiking* and *turnandopen* domains. The use of delay checks (absent in $\mathcal{C}_{\neg DelCk}$) and of refinement variables (absent in $\mathcal{C}_{\neg DecVars}$) is less critical but contributes to the scaling up of the planners on some hierarchical domains. The gain of using causal networks (absent in $\mathcal{C}_{\neg CN}$) and A^ϵ (absent in $\mathcal{C}_{A^*}$) is also globally noticeable and benefits many domains while not being critical to any of them.

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

	$\mathcal{C}_{full}$	$\mathcal{C}_{gen}$	$\mathcal{C}_{lift}$	$\mathcal{C}_{A^*}$	$\mathcal{C}_{\neg DelCk}$	$\mathcal{C}_{\neg DecVars}$	$\mathcal{C}_{\neg CN}$
(IPC4) airport-GEN	**6**	**6**	5	**6**	**6**	**6**	5
(IPC4) airport-tw-GEN	**7**	**7**	6	**7**	**7**	**7**	6
(IPC2) blocks-GEN	25	25	**26**	23	25	25	24
(LAAS) dock-worker-GEN	**9**	**9**	**9**	**9**	**9**	**9**	**9**
(IPC8) driverlog-GEN	**4**	**4**	1	0	**4**	**4**	1
(LAAS) handover-GEN	**4**	**4**	3	**4**	**4**	**4**	**4**
(IPC2) logistics-GEN	**23**	**23**	4	12	**23**	**23**	7
(IPC4) pipesworld-dl-GEN	**6**	**6**	**6**	**6**	**6**	**6**	5
(IPC5) rovers-GEN	34	34	**38**	32	34	34	33
(IPC4) satellite-tw-GEN	**10**	**10**	**10**	7	**10**	**10**	8
(IPC8) satellite-GEN	16	16	**17**	12	16	16	14
Total Generative	**144**	**144**	125	118	**144**	**144**	116
(IPC2) blocks-FULLHIER	10	7	**15**	10	10	4	10
(IPC2) blocks-PARTHIER	27	27	**32**	26	27	29	28
(LAAS) dock-worker-FULLHIER	**22**	**22**	**22**	**22**	14	20	**22**
(LAAS) dock-worker-PARTHIER	**12**	9	**12**	**12**	**12**	7	**12**
(LAAS) handover-FULLHIER	**10**	**10**	1	**10**	**10**	**10**	**10**
(IPC8) hiking-FULLHIER	**20**	2	2	**20**	17	12	**20**
(IPC2) logistics-PARTHIER	**28**	**28**	27	27	**28**	**28**	**28**
(LAAS) race-FULLHIER	**13**	10	5	**13**	**13**	**13**	**13**
(IPC8) satellite-PARTHIER	**18**	**18**	**18**	14	**18**	**18**	14
(LAAS) baking-PARTHIER	**6**	**6**	**6**	**6**	**6**	**6**	5
(IPC8) turnandopen-FULLHIER	**8**	0	0	**8**	**8**	0	**8**
Total Hierarchical	**174**	139	140	168	163	147	170
Total	**318**	283	265	286	307	291	286

Table 4: Number of problems solved by each configuration with a 30 minutes timeout.

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 witnessed in Figure 13, where we can see that the lifted version tends to solve simple problems faster because it does not need to ground the problem.

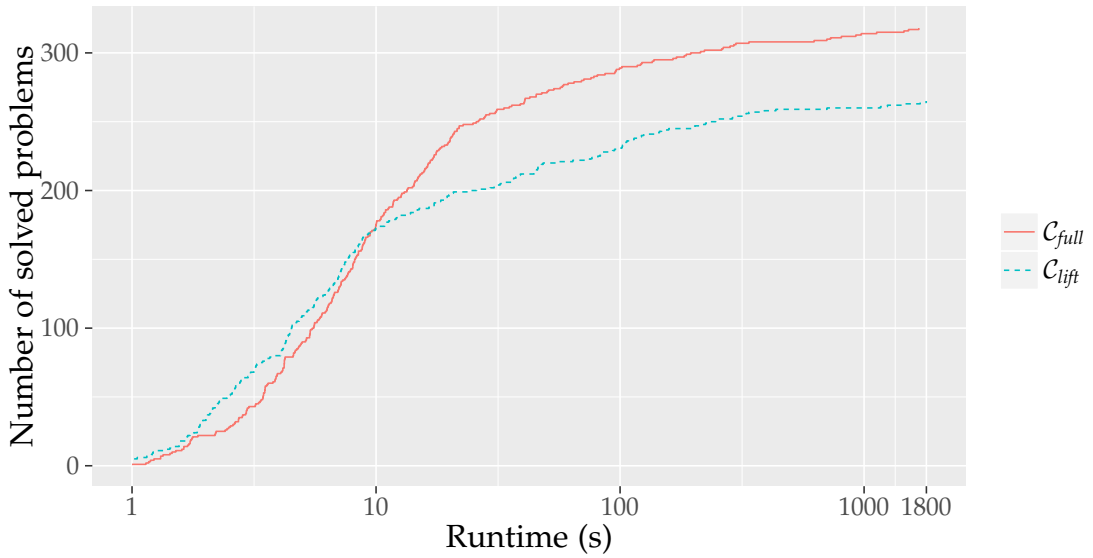


Figure 13: Time needed to solve problems with $\mathcal{C}_{full}$ and $\mathcal{C}_{lift}$. The Java Virtual Machine takes an important share of time to load and perform optimization during the first seconds of the run. This overhead is however comparable for both configurations.

Table 5 gives the average number of nodes expanded by each configuration on problems solved by all configurations. On those problems, it should be apparent that the full configuration of FAPE actually requires very little search to find solutions. Other configurations, and $\mathcal{C}_{lift}$ especially, tend to explore a larger part of the search space. On all these problems, the average branching factor is between 2 and 3.5, with only small variations between the different configurations.

Partial vs Full Hierarchy. A fair number of domains use partial hierarchies. A simple and relevant example of the benefits of using partial hierarchies is the *blocks*-PartHier domain, given in Appendix B.2. 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

	$\mathcal{C}_{full}$	$\mathcal{C}_{gen}$	$\mathcal{C}_{lift}$	$\mathcal{C}_{A^*}$	$\mathcal{C}_{\neg DelCk}$	$\mathcal{C}_{\neg DecVars}$	$\mathcal{C}_{\neg CN}$
(IPC4) airport-GEN	164	164	226	**154**	164	164	314
(IPC4) airport-tw-GEN	204	204	284	**195**	204	204	450
(IPC2) blocks-GEN	162	162	201	**139**	162	162	182
(LAAS) dock-worker-GEN	90	90	**85**	152	90	90	90
(LAAS) handover-GEN	**17**	**17**	15 931	25	**17**	**17**	73
(IPC2) logistics-GEN	**26**	**26**	24 016	36	**26**	**26**	3 440
(IPC4) pipesworld-dl-GEN	**230**	**230**	898	612	**230**	**230**	1 436
(IPC5) rovers-GEN	105	105	**98**	153	105	105	104
(IPC4) satellite-tw-GEN	50	50	**46**	1 824	50	50	676
(IPC8) satellite-GEN	**125**	**125**	292	230	**125**	**125**	184
Total Generative	**1 174**	**1 174**	42 076	3 520	**1 174**	**1 174**	6 949
(IPC2) blocks-FULLHIER	62	263	**39**	62	88	72	70
(IPC2) blocks-PARTHIER	84	84	**76**	121	84	89	566
(LAAS) dock-worker-FULLHIER	38	**33**	631	38	41	121	39
(LAAS) dock-worker-PARTHIER	**90**	498	101	**90**	93	75 088	**90**
(LAAS) handover-FULLHIER	**12**	**12**	163	**12**	**12**	16	**12**
(IPC8) hiking-FULLHIER	**34**	36	61	**34**	236	55	**34**
(IPC2) logistics-PARTHIER	**50**	**50**	56	50	50	84	516
(LAAS) race-FULLHIER	**17**	17	376	**17**	32	29	**17**
(IPC8) satellite-PARTHIER	**87**	**87**	112	403	**87**	110	105
(LAAS) baking-PARTHIER	**58**	847	**58**	**58**	**58**	**58**	243
Total Hierarchical	**531**	1 925	1 675	885	781	75 723	1 691
Total	**1 705**	3 099	43 751	4 405	1 955	76 897	8 640

Table 5: Average number of expanded partial plans on problems solved by all configurations. This number does not integrate flaws with a single resolver as FAPE would resolve any such flaw immediately, before adding the node to the search tree.

the other hand, it can use as many pickup, putdown and unstack actions as necessary to establish the conditions of the stack actions. This formulation puts a constraint on the plan: do not use more stack actions than necessary. As result, the planner solves more instances of the partially hierarchical domain and does it on average 3.7 times faster than for the generative version. When compared to the fully hierarchical version of the domain (Appendix B.3), the partially hierarchical one is obviously simpler and requires 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 performs well.

A similar approach is taken in the *handover*-PARTHIER and *logistics*-PARTHIER. In *handover*, the only task-dependent actions are the ones involving manipulation of an object (pick, place and handover) while the movements of the robots are left free. Similarly, in *logistics* the only hierarchical parts are the load and unload actions. The movements of the fleet of planes and trucks are left free. If the planner was to entirely decompose 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.

5.2 Empirical Comparison with IPC Planners

Experimental setup. For comparison with state of the art PDDL temporal planners, we consider the 11 temporal domains from the International Planning Competition (IPC) all of which have 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-dependent 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. All domains, in their ANML and PDDL versions, are available online in FAPE's public repository.[5]

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 h^{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 [EMR12]) 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 were runner-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 is therefore strictly less expressive than the other planners considered here.

We distinguish two versions of FAPE. FAPE-Gen denotes the purely generative version of FAPE that only considered generative domain encodings 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 generative domains, it uses the general search strategy and is equivalent to FAPE-Gen.

Results. All tests were performed on an Intel Core i7 with 3GB of RAM and allowed to run for 30 minutes. The results are given in Table 6 in terms of the number of problems solved by each planner within the time limit.

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

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, but still lags behind TFD in number of solved problems. However it should be noted that the advantage of TFD can be entirely reduced to its excellent performance on the *airport* domain. This domain is particularly problematic for FAPE because it contains conditional effects that are not natively supported by FAPE and that were worked around by partially grounding the problem during the translation to ANML. This forced grounding however interacts badly with FAPE' search strategy that relies on lifted exploration and very negatively impacts its performance on this particular domain.

	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	6	**13**	2
(ipc5) rovers	**34**	**34**	26	26	29
(ipc4) satellite-tw	**10**	**10**	6	4	0
(ipc8) satellite	16	16	3	4	**17**
(ipc8) turnandopen*	0	8	8	9	**18**
(ipc8) hiking*	0	**20**	10	9	19
Total	130	165	142	138	**185**

Table 6: 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.

A focused subset of the results is given in Table 7 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 counterpart. While being the overall best performer, TFD exhibits poor performance on those domains, solving only 3 problems.

	FAPE (Gen/Hier)	POPF	OPTIC	TFD
(ipc4) airport-tw-Gen	**17**	7	**7**	1
(ipc4) pipesworld-dl-Gen	6	6	**13**	2
(ipc4) satellite-tw-Gen	**10**	6	4	0
Total	23	19	24	3

Table 7: Results limited to domains featuring deadlines or timewindows. FAPE-Hier does not appear separately as we only considered generative versions of those domains.

6 Related work and discussion

6.1 PDDL Temporal Planners

STRIPS and PDDL. The original PDDL language [McD+98] and its ancestors STRIPS and ADL, define actions as state-transition functions with a uniform duration. This was extended with the introduction of PDDL2.1 for the purpose of the third International Planning Competition (IPC) [FL03]. 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 the end times of the action as well as durative conditions that require some condition to hold in all states traversed while performing the action. While this seems like a minor extension, it allows the expression of temporal planning problems with required concurrency [Cus+07a] and with interdependent actions [CMR13].

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. While this limitation can be avoided by splitting a complex temporal action with intermediate time points into multiple subactions [FLH04], encoding such durative actions by hand is difficult and error prone. A possible approach is to compile a more expressive language into PDDL2.1 to benefit from its large ecosystem (as done for PDDL-TE [CMR10] or for ANML [SFC08]).

The PDDL2.1 language has been further extended with *timed initial literals* that allow truth assignments on predicates at arbitrary times (in PDDL2.2 [EH04]) and temporally extended goals expressed as constraints on the state trajectory (in PDDL3.0 [GL05]). Even though those extensions are essential in representing real world problems, they have not been much used in temporal planning, e.g., none of the participants of the temporal track in the IPC-14 supported them natively.[6]

Forward-chaining planners. Not surprisingly, most temporal planners participating in planning competitions evolved from classical planners. Similarly to the classical planning tracks , the temporal satisficing tracks have been dominated by forward-chaining temporal planners. We can partition forward-search planners into three categories depending on their search space:

First-fit temporal planners are essentially classical planners that temporally schedule a sequential solution. The most surprising example is the baseline planner that (unofficially) won the temporal track of IPC-2008 by greedily rescheduling the sequential solution provided by MetricFF [Hof03]. A more advanced implementation of this approach is YAHSP [Vid04; Vid11; Vid14] whose second and third versions respectively won the temporal tracks of IPC-2011 and IPC-2014. 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].

Decision-epoch planners maintain 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], and others [HG01; Has06; BK00]. These planners support

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

some cases of required concurrency but are still not complete for temporally expressive problems [Cus+07a].

Temporally lifted planners separate the problems of what actions to add to the plan and when to schedule them by using an STN to keep track of temporal constraints. These ideas have been first introduced by CRIKEY [HLF04] and have been developed in its successors: 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. Most of the heuristics are based on a *temporal relaxed planning graph* (TRPG) built by ignoring the delete effects of actions. These heuristics are generally adapted from the one that have been successful in classical planning such as h^{add} [BG01], h^{cea} [HG08] or h^{FF} [HN01].

Other notable approaches. The GraphPlan framework [BF97] has seen many extensions to handle temporal planning, as first demonstrated by Smith and Weld [SW99] whose planner finds a solution by extracting it from a planning graph with temporally annotated nodes. In the same line, LPGP [LF03] and TLP-GP [MR08] both decouple the causal parts of the problem, dealt with in a GraphPlan framework, and the temporal parts that are addressed by a linear programming solver or a disjunctive temporal network.

LPG [GSS03; GSS06] builds an action graph (with similarities to planning graphs) through stochastic local search. Its latest version is able to handle problems with required concurrency by splitting durative actions into instantaneous ones while considering temporal constraints in an STN [GSS10].

Temporal plan-space planning has been represented by VHPOP [YS03], a ground plan-space planner that uses an adaptation of the 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 seeks minimal makespan plans. 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. The inference in CPT relies on dedicated pruning rules, handled as a constraint satisfaction problem.

ITSAT [RGS15] is the first satisfiability-based planner to support PDDL2.1 problems with required concurrency. This is achieved by first solving, with a SAT solver, an atemporal 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 additional clauses forbiding the cause of the temporal inconsistency and the procedure is restarted.

6.2 Hierarchical planners

HTN planning has initially been developed around a plan-space approach with planners such as NOAH [Sac75], Nonlin [MR91], SIPE [Wil90; WM95], O-Plan [TDK94] and UMCP [EHN94]. For totally ordered HTNs, it is possible to plan tasks in order they will be executed, and thus, to knows the current world state at each step of the planning process. This was proposed in SHOP [Nau+00], and extended in SHOP2 [Nau+03] for partially ordered tasks and durative actions. A Multi-Timeline Preprocessing technique is proposed to translate PDDL2.1 operators such as to keep track of temporal information in the current state. Each operator is augmented with start

time, duration and read-time and write-time primitives for time bookkeeping upon instantiation. A more general procedure to transform durative actions in HTN methods is proposed in [Gol06]: it maps an action into a task network with two or three operators, one for the start of the durative action, one for its end, and an epsilon length "spacer" pseudo action. It also adds a time fluent and proposes a modification of the planner heuristic to handle these extensions. [YS05] proposes to extend HTN planning with temporal propagation on associated 'local' STNs; these are limited to adjacent nodes in the task network, and thus much smaller than a global STN.

SIADEX [Cas+06; FO+06] is an elaborate temporal state-based HTN planner. It 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. However, it also performs an online scheduling of the plan by constraining an action to start after all its preconditions are true (using an STN to keep track of temporal constraints). XEPlanner [Tan+12] is an HTN planner designed specifically for addressing emergencies in dynamic situations. It supports durative actions, temporally-enhanced methods and axioms. Planning is done with an anytime heuristic algorithm handling the task network together with an STN. GSCCB-SHOP2 [Qi+17] extends SHOP2 for handling time and resources. Specific state-updating rules are used for resource reasoning, together with a backtrack consistency checking for managing simple temporal constraints.

HTN Planning with task insertion extends the traditional HTN formalism by allowing the use of primitive actions at arbitrary places in the solution plan [GB11]. It was first explored with PANDA, a lifted hierarchical planner reasoning in plan space [Sch09]. PANDA allows the use of high-level actions that can be decomposed into lower-level ones. Both high-level and primitive actions can 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 to account for causal and hierarchical features in a best first search setting [Sch09; Elk+12; BKB14]. Their effect on the scalability of PANDA is however unconvincing and does not allow the planner to handle complex plans.[7] HTN planning with task insertion is also supported by HiPOP, a ground hierarchical planner reasoning 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.

The HTN planning approach of [SBM09] handles qualitative preferences on actions together with temporally extended state preferences and constraints (as in PDDL3), expressed in a subset of LTL. Time *per se* is not explicit in the representation. The modal operators are compiled out at preprocessing time into additional predicates. A Branch&Bound algorithm, benefitting from HTN methods and pruning heuristics, is used to find a most preferable plan. The approach is used in [Soh+13] for the automated composition of software components, as in the composition of web services and stream processing systems.

Most HTN planners do not rely on heuristic guidance or reachability analysis but instead perform a depth search guided by on the domain-specific control knowledge provided by their hierarchical methods. Interest in using the delete-relaxation for hierarchical planning has been studied theoretically [Alf+14] and in particular with temporal domains in a preliminary version

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

of our work [BM16]. Heuristics for non-temporal HTN have have been explored in the context of the plan-space hierarchical planner PANDA where no state-information is available for method selection [Sch09; BKB14] as well as in forward progression search [Höl+20; HBB20].

6.3 Timeline based Representations and Planners

Timeline-based representations focus on scheduling various temporal intervals representing values taken by state variables. In these approaches, actions are typically composed of a set of temporally qualified assertions representing the action conditions and effects over various state variables (or 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 (e.g. [Chi+00b; TDK94; DU11; Ces+09; EkR96; FJ03; Bar+12; Mus+02]), no dominating language has emerged for the encoding of such planning problems.

While much work on temporal planning can be seen an incremental evolution from classical planning, research 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, led to numerous planners with advanced temporal representation capabilities [Ver83; GL94; Mus94; PW94; FJ03]. IxTeT [GL94] is a notable least-commitment planner that allows reasoning on time and resources in a plan-space approach. A large part of the internal representation and reasoning is handled by specific constraints 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.

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. The planner's objective is to find fully defined timelines that respect all compatibilities. HSTS was notably used for the Remote Agent Planner (RAX-PS) that was demonstrated on board for controlling the operations of the Deep Space One spacecraft [Nay+99]. HSTS matured into EUROPA [FJ03; Bar+12], and its language NDDL, 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. EUROPA relies on domain-dependent knowledge to guide a depth-first search. Efforts to transpose domain-independent heuristics into EUROPA but have seen limited results [BS07; BS08].

The European Space Agency 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. It has been used as a building block for the OMPS [CFP08] and GOAC-APSI [Fra+11] planners. Both use a search algorithm similar to EUROPA. APSI has also been used for MrSPOCK [Ces+09], a long term planner for the Mars Express mission that works by greedily constructing a long term plan optimized with a genetic algorithm.

A similar approach is taken in the meta-CSP framework which addresses a planning and scheduling problem as a higher-level constraint satisfaction problem that requires cross reason-

ing on several lower level CSPs (referred to as ground CSPs). Meta-constraints enforce high-level requirements on the solution plan, playing the role of the flaw detection functions of other timeline based planners. The detected flaws are handled by posting additional constraints on the ground CSPs. The key idea is to permit an easy integration of application specific components by the addition of supplementary ground CSPs and meta-constraints. The meta-CSP framework has recently been used in CHIMP, a planner with a timeline based representation [Sto+15]. Like other HTN planners, CHIMP uses a notion of task that can be decomposed into partially ordered task through the application of methods.

ASPEN is another timeline based planner [Fuk+97; Smi+98; Chi+00a; Chi+00b], which uses the AML language [Fuk+97; Chi+00b]. Its timelines parameters are handled in a dedicated constraint network. It uses an iterative repair technique [Zwe+93] to perform local search in place of the depth-first search adopted by other timeline based planners. Efficiency is sought by the definition of a hierarchical structure where activities can be refined into sub-activities, allowing the planner to quickly bootstrap its search with a minimal (possibly flawed) plan.

The Action Notation Modeling Language (ANML [SFC08]) is a proposal to overcome the absence of apparent causal structure of NDDL, the limited support for generative planning of AML and the lack of hierarchy in IxTeT. 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. In addition, ANML provides some facilities for hierarchical planning: 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 and can be used to express subtasks. 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. The original language definition can be characterized as HTN with task-insertion [Alf+16]: additional actions can be placed at arbitrary places in the solution plan. The latest version of ANML proposes to restrict the possible placement of actions relatively to higher-level actions by marking them as *motivated* [Sch+13]. Conceptually, the presence of such an action must be "motivated" by the presence of an higher level action that requires its presence and temporally envelops it and is closely related to our *task-dependency* concept. To the best of our knowledge no planner exists that support all ANML features and in particular, none that supports any of its hierarchical features. TAMER [VMC20] is a forward-chaining planner that supports a form of intermediate conditions and effects of the ANML language. More specifically, it supports conditions and effects constrained to be a fixed time-amount after the start or before the end of an action. LCP [Bit18] supports a more complete set of temporal features, equivalent to the ones in FAPE, by constructing a sequence of scheduling problems that are solved with an SMT solver. While we are aware of other prototypes of ANML planners (at NASA Ames, Adventium Lab and Fundazione Bruno Kessler) none of them have been the subject of a publication nor go as far as FAPE in the support of temporal and hierarchical features. FAPE is thus the first planner to support most of the expressive temporal and hierarchical features of the ANML modeling language.

7 Conclusion

We reported here on FAPE, a Flexible Acting and Planning Environment based on timelines. To our knowledge, FAPE is the first planner supporting both the temporal and hierarchical fea-

tures of the expressive ANML modeling language. ANML has significant advantages because it consistently blends flexible timelines with hierarchical refinement methods, when available. We presented a planning algorithm for the proposed representation, discussed the specifics of its search space, and proved its soundness and completeness. A significant contribution of the presented work is the development of well informed heuristics and inference methods for this algorithm. The approach takes into account an original reachability analysis supporting causal networks that are explicitly maintained by the planner and used to focus the search. A comprehensive experimental evaluation, using the standard benchmarks of the field, allowed us to assess several search strategies for FAPE and to compare its performance to other planners, with or without hierarchical decomposition knowledge. Our evaluation reflects that the proposed techniques for this expressive representation are competitive.

Furthermore, there is certainly room for a number of improvements of our techniques and implementation, and a large opportunity for optimizing planning domain knowledge, which we have not yet explored. Since the source code of the planner and all the domains presented here are openly available, we do hope that this article will generate interest in the temporal planning community and trigger efforts to address these topics and extensions. An important extension that deserves to be further investigated is the support for resources, for which many techniques have been devised by the constraint-based planning and scheduling community [Vil07].

A strong motivation for the development of FAPE is to support the integration of planning and acting. The latter involves opportunistically instantiating the unbound variables remaining in a synthesized plan, in particular for temporal variables through a dispatching algorithm, and refining planned actions into executable commands. The integration also requires plan repair techniques, and the assessment of when replanning is preferable to repairing. These developments, not presented here, have been addressed and integrated within the design of an activity manager interleaving planning and acting [BM16, Chap. 5]. Experiments with a PR2 robot and in simulation indicates that the proposed representation is very convenient for handling a temporally rich domain at the planning as well as the acting levels.

In a temporally rich domain, an actor has to relate its actions to exogenous events, which is feasible with known techniques when the occurrence of these events is fully observable [Mor14]. In many application areas, and in service robotics in particular, full observability is not a realistic assumption. One has to check whether a plan is dynamically controllable despite partial observability, and if not, to decide what needs to be observed to make it controllable, and determine how to consistently integrate the required sensing actions with other planned activities. The proposed approach turned out to be quite convenient for supporting the corresponding developments within FAPE [BMGI16].

For the sake of space, this paper does not cover the two issues of acting and partial observability. However, they need to be mentioned as they open several promising avenues for future investigations and developments using the ANML representation.

References

[AK83] J. Allen and J. A. Koomen. "Planning Using a Temporal World Model". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1983.

[Alf+14] R. Alford, V. Shivashankar, U. Kuter, and D. S. Nau. "On the Feasibility of Planning Graph Style Heuristics for HTN Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2014.

[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. 2016.

[All83] J. Allen. "Maintaining Knowledge About Temporal Intervals". In: *Communications of the ACM* 26.11 (1983).

[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.

[BCC12] J. Benton, A. Coles, and A. Coles. "Temporal Planning with Preferences and Time-Dependent Continuous Costs". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2012.

[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.

[Bec16] P. Bechon. "Planification multirobot pour des missions de surveillance avec contraintes de communication". PhD thesis. Université de Toulouse, 2016.

[BF97] A. L. Blum and M. L. Furst. "Fast Planning through Planning Graph Analysis". In: *Artificial Intelligence* 90.1–2 (1997).

[BG01] B. Bonet and H. Geffner. "Planning as Heuristic Search". In: *Artificial Intelligence* 129.1-2 (2001).

[Bit18] A. Bit-Monnot. "A Constraint-Based Encoding for Domain-Independent Temporal Planning". In: *International Conference on Principles and Practice of Constraint Programming (CP)*. 2018.

[BK00] F. Bacchus and F. Kabanza. "Using Temporal Logics to Express Search Control Knowledge for Planning". In: *Artificial Intelligence* 116.1-2 (2000).

[BKB14] P. Bercher, S. Keen, and S. Biundo. "Hybrid Planning Heuristics Based on Task Decomposition Graphs". In: *International Symposium on Combinatorial Search (SoCS)*. 2014.

[BM16] A. Bit-Monnot. "Temporal and Hierarchical Models for Planning and Acting in Robotics". PhD thesis. Université de Toulouse, 2016.

[BMGI16] 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)*. 2016.

[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.

[BS08] S. Bernardini and D. E. Smith. "Automatically Generated Heuristic Guidance for EUROPA2". In: *International Symposium on AI, Robotics, and Automation in Space (iSAIRAS)*. 2008.

[Cas+06] L. A. Castillo, J. Fdez-Olivares, O. García-Pérez, and F. Palao. "Efficiently Handling Temporal Knowledge in an HTN Planner". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2006.

[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.

[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).

[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.

[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.

[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)*. 2010.

[CMR13] M. C. Cooper, F. Maris, and P. Régnier. "Managing Temporal Cycles in Planning Problems Requiring Concurrency". In: *Computational Intelligence* 29.1 (2013).

[Col+08] A. Coles, M. Fox, D. Long, and A. Smith. "Planning with Problems Requiring Temporal Coordination". In: *AAAI Conference on Artificial Intelligence*. 2008.

[Col+10] A. Coles, A. Coles, M. Fox, and D. Long. "Forward-Chaining Partial-Order Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2010.

[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).

[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.

[Cus+07b] W. Cushing, S. Kambhampati, K. Talamadupula, D. S. Weld, and Mausam. "Evaluating Temporal Planning Domains". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2007.

[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).

[DMP91] R. Dechter, I. Meiri, and J. Pearl. "Temporal Constraint Networks". In: *Artificial Intelligence* 49 (1991).

[DU11] M. B. Do and S. Uckun. "Timeline-based Planning System for Manufacturing Applications". In: *IJCAI Workshop on Artificial Intelligence and Logistics (AILog)*. 2011.

[Dvo+14] 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)*. 2014.

[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.

[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.

[EkR96] A. El-kholy and B. Richards. "Temporal and Resource Reasoning in Planning: the parcPLAN approach". In: *European Conference on Artificial Intelligence (ECAI)*. 1996.

[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.

[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 (STAR)* 76 (2012).

[FJ03] J. Frank and A. Jónsson. "Constraint-Based Attribute and Interval Planning". In: *Constraints* 8.4 (2003).

[FL03] M. Fox and D. Long. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003).

[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.

[Flo62] R. W. Floyd. "Algorithm 97: Shortest Path". In: *Communications of the ACM* 5.6 (1962).

[FO+06] J. Fdez-Olivares, L. A. Castillo, O. García-Pérez, and F. Palao. "Bringing Users and Planning Technology Together. Experiences in SIADEX". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2006.

[Fra+11] S. Fratini, A. Cesta, R. De Benedictis, A. Orlandini, and R. Rasconi. "APSI-based Deliberation in Goal Oriented Autonomous Controllers". In: *Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*. 2011.

[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: *International Symposium on AI, Robotics, and Automation in Space (iSAIRAS)*. 1997.

[GA83] M. Ghallab and D. G. Allard. "A-epsilon - An Efficient Near Admissible Heuristic Search Algorithm". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1983.

[GB11] T. Geier and P. Bercher. "On the Decidability of HTN Planning with Task Insertion". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2011.

[GL05] A. Gerevini and D. Long. *Plan Constraints and Preferences in PDDL3*. Tech. rep. Department of Electronics for Automation, University of Brescia, 2005.

[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.

[GNT04] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. 2004.

[GNT16] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[Gol06] R. P. Goldman. "Durative Planning in HTNs". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2006.

[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).

[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).

[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 Scheduling (ICAPS)*. 2010.

[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).

[HBB20] D. Höller, P. Bercher, and G. Behnke. "Delete- and Ordering-Relaxation Heuristics for HTN Planning". In: *Internation Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by C. Bessiere. ijcai.org, 2020, pp. 4076–4083. DOI: 10.24963/ijcai.2020/564.

[HG01] P. Haslum and H. Geffner. “Heuristic Planning with Time and Resources”. In: *European Conference on Planning (ECP)*. 2001.

[HG08] M. Helmert and H. Geffner. “Unifying the Causal Graph and Additive Heuristics”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2008.

[HLF04] K. Halsey, D. Long, and M. Fox. “CRIKEY - A Temporal Planner Looking at the Integration of Scheduling and Planning”. In: *ICAPS Workshop on Integrating Planning into Scheduling*. 2004.

[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).

[Hof01] J. Hoffmann. “FF: The Fast-Forward Planning System”. In: *AI Magazine* 22.3 (2001).

[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).

[Höl+20] D. Höller, P. Bercher, G. Behnke, and S. Biundo. “HTN Planning as Heuristic Progression Search”. In: *Journal of Artificial Intelligence Research (JAIR)* 67 (2020), pp. 835–880. DOI: `10.1613/jair.1.11282`.

[Jon+00] A. K. Jonsson, P. H. Morris, N. Muscettola, K. Rajan, and B. Smith. “Planning in Interplanetary Space: Theory and practice”. In: *International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*. 2000.

[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* 76.1-2 (1995).

[Kno94] C. A. Knoblock. “Automatically Generating Abstractions for Planning”. In: *Artificial Intelligence* 68.2 (1994).

[LF03] D. Long and M. Fox. “Exploiting a Graphplan Framework in Temporal Planning”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2003.

[Mac77] A. K. Mackworth. “Consistency in Networks of Relations”. In: *Artificial Intelligence* 8.1 (1977).

[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.

[Mor14] P. H. Morris. “Dynamic Controllability and Dispatchability Relationships”. In: *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*. 2014.

[MR08] F. Maris and P. Régnier. “TLP-GP: Solving Temporally-expressive Planning Problems”. In: *International Symposium on Temporal Representation and Reasoning (TIME)*. 2008.

[MR91] D. McAllester and D. Rosenblitt. “Systematic Nonlinear Planning”. In: *AAAI Conference on Artificial Intelligence*. 1991.

[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*. 2002.

[Mus94] N. Muscettola. “HSTS: Integrating Planning and Scheduling”. In: *Intelligent Scheduling* (1994).

[Nau+00] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. *SHOP and M-SHOP: Planning with Ordered Task Decomposition*. Tech. rep. CS TR 4157, University of Maryland, 2000.

[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).

[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: *International Symposium on AI, Robotics, and Automation in Space (iSAIRAS)*. 1999.

[Pla08] L. Planken. "New Algorithms for the Simple Temporal Problem". Master Thesis. Delft University, 2008.

[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.

[PW94] J. S. Penberthy and D. S. Weld. "Temporal Planning with Continuous Change". In: *AAAI Conference on Artificial Intelligence*. 1994.

[Qi+17] C. Qi, D. Wang, H. Muñoz-Avila, P. Zhao, and H. Wang. "Hierarchical task network planning with resources and temporal constraints". English. In: *Knowledge-Based Systems* 133 (Oct. 2017), pp. 17–32. DOI: 10.1016/j.knosys.2017.06.036. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950705117303167.

[Rab+99] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee. "Iterative Repair Planning for Spacecraft operations using the ASPEN system". In: *International Symposium on AI, Robotics, and Automation in Space (iSAIRAS)*. 1999.

[RGS15] M. F. Rankooh and G. Ghassem-Sani. "ITSAT: An Efficient SAT-Based Temporal Planner". In: *Journal of Artificial Intelligence Research (JAIR)* 53 (2015).

[Sac75] E. D. Sacerdoti. "The Nonlinear Nature of Plans". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1975.

[SBM09] S. Sohrabi, J. A. Baier, and S. A. McIlraith. "HTN Planning with Preferences." In: *IJCAI* (2009).

[Sch09] B. Schattenberg. "Hybrid Planning & Scheduling". PhD thesis. Ulm University, 2009.

[Sch+13] A. Schwerdfeger, M. Boddy, D. E. Smith, J. Frank, and C. Mcgann. *The ANML Language, Revised [DRAFT]*. Tech. rep. Adventium Labs, 2013.

[SFC08] D. E. Smith, J. Frank, and W. Cushing. "The ANML Language". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2008.

[Smi03] D. E. Smith. "The Case for Durative Actions: A Commentary on PDDL2.1." In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003).

[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).

[Soh+13] S. Sohrabi, O. Udrea, A. Ranganathan, and A. Riabov. "HTN Planning for the Composition of Stream Processing Applications." In: *ICAPS* (2013).

[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)*. 2015.

[SW99] D. E. Smith and D. S. Weld. "Temporal Planning with Mutual Exclusion Reasoning." In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 1999.

[Tan+12] P. Tang, H. Wang, C. Qi, and J. Wang. "Anytime heuristic search in temporal HTN planning for developing incident action plans". In: *AI Communications* 25.4 (Sept. 2012), pp. 321–342. DOI: 10.3233/AIC-2012-0539. URL: https://twin.sci-hub.tw/6191/84c8746f47d76132b2bba4d6301867e3/10.3233@AIC-2012-0539.pdf.

[TDK94] A. Tate, B. Drabble, and R. Kirby. "O-Plan2: An Architecture for Command, Planning and Control". In: *Intelligent Scheduling* (1994).

[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 (1983).

[VG06] V. Vidal and H. Geffner. "Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming". In: *Artificial Intelligence* 170.3 (2006).

[Vid04] V. Vidal. "A Lookahead Strategy for Heuristic Search Planning". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. 2004.

[Vid11] V. Vidal. "YAHSP2: Keep It Simple, Stupid". In: *International Planning Competition (IPC)*. 2011.

[Vid14] V. Vidal. "YAHSP3 and YAHSP3-MT in the 8th International Planning Competition". In: *International Planning Competition (IPC)*. 2014.

[Vil07] P. Vilím. "Global Constraints in Scheduling". PhD thesis. Charles University in Prague, Faculty of Mathematics, Physics, Department of Theoretical Computer Science, and Mathematical Logic, 2007.

[VMC20] A. Valentini, A. Micheli, and A. Cimatti. "Temporal Planning with Intermediate Conditions and Effects." In: *AAAI Conference on Artificial Intelligence*. 2020.

[War62] S. Warshall. "A Theorem on Boolean Matrices". In: *Journal of the ACM* 8.1 (1962).

[Wil+08] D. E. Wilkins, S. F. Smith, L. A. Kramer, T. J. Lee, and T. W. Rauenbusch. "Airlift mission monitoring and dynamic rescheduling". English. In: *Engineering Applications of Artificial Intelligence* 21.2 (Mar. 2008), pp. 141–155. DOI: `10.1016/j.engappai.2007.04.001`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0952197607000486`.

[Wil90] D. E. Wilkins. "Can an AI Planner Solve Practical Problems?" In: *Computational Intelligence* 6.4 (1990).

[WM95] D. E. Wilkins and K. Myers. "A Common Knowledge Representation for Plan Generation and Reactive Execution". In: *Journal of Logic and Computation* 5.6 (1995).

[YS03] H. L. S. Younes and R. G. Simmons. "VHPOP: Versatile Heuristic Partial Order Planner". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003).

[YS05] N. Yorke-Smith. "Exploiting the Structure of Hierarchical Plans in Temporal Constraint Propagation." In: *AAAI Conference on Artificial Intelligence*. 2005.

[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).

A Overlength Proofs

A.1 Proof of Soundness and Completeness (Proposition 3.2)

Proposition 3.2. FAPE *is sound and complete.*

Proof.
Soundness. Soundness requires proving that any plan returned by FAPEPLAN is indeed a solution, i.e., that it respects the four conditions of Definition 2.4.

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 refinement, 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.

Each of the last three conditions corresponds 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.

Completeness. We rely on the study by Schattenberg [Sch09, Sec. 2.6 to 2.8] and Kambhampati, Knoblock, and Yang [KKY95] 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 resolvers associated with 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].

We first show that no solution plan is ruled out because of the presence of a flaw. We assume that a partial plan ϕ has a flaw of a given type and show that it cannot be a solution plan or that it will be transformed into an equivalent plan.

- if ϕ has an unrefined task τ then it is not a solution plan according to Definition 2.4.
- 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 a 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 to their state variable during $[end(\beta), start(\alpha)]$. Even though this situation could trigger a flaw, the 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 requirement for causal support, that all assertions in a solution plan must have an incoming causal link (or an equivalent set of persistence assertions) 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 will 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 an option 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 derived from an existing task. The former is handled by the *task supporter* resolvers that allow choosing any of the future assertions derived from an existing task. For the latter, let us observe that the containing action will be part of another refinement tree not yet in the plan, whose root is necessarily a free action. Our *free action* resolvers allow the planner 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 consideration of all possible sources of supporting assertions, namely: *(i)* those already in the plan, *(ii)* those introduced by the extension of existing refinement trees, and *(iii)* those introduced by the extension of new refinement trees.
- **Unrefined task.** Assuming that no previous commitment was made to 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, the possibility of using A to refine τ will be considered in other branches of the search space: those derived 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 to their state variables are equal, and *(iii)* 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 therefore complete.

□

A.2 Proof of the convergence of earliest appearances (Proposition 4.1)

Proposition 4.1. *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. We first suppose that all actions have a single condition. With this assumption, there are two sets of trivially reachable elements: fluents in initial timed literals (i.e. in I) 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. 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 be 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 the set of truly reachable elements $E^{TrulyReach}$ which is a subset of the set of assumed reachable elements $E^{AssReach}$. We define as $E^{NotTrulyReach}$ the subset of $E^{AssReach}$ that is not reachable.

An element d is reachable if there exists 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 $E^{TrulyReach} \cup E^{NotTrulyReach}$. To show that earliest appearances of reachable nodes eventually converge to their true values, we first show that the earliest appearances of nodes in $E^{NotTrulyReach}$ indefinitely increase until they are removed from the model.

A node $n \in E^{NotTrulyReach}$ 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 no path from 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 $E^{NotTrulyReach}$ provides the earliest start time of p:

- $ea(p) = ea(n) + spp(n, p)$. In this case, this is an infeasible causal loop involving n and the earliest appearance of n will be increased at each iteration.
- there is another node $n' \in E^{NotTrulyReach}$ such that $ea(p) = ea(n') + spp(n', p)$. In this case, we can recursively perform similar reasoning on n': it is either part of an infeasible causal loop or depends on a node $n'' \in E^{NotTrulyReach} \setminus \{n, p\}$. 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 $E^{NotTrulyReach}$ indefinitely increase until they are removed. This is also the case for all unreachable nodes that were once assumed reachable because the sources of all their shortest paths is a node in $E^{NotTrulyReach}$.

On the other hand, we will eventually reach a point where:

$$\min_{s \in E^{TrulyReach}} ea(s) + spp(s, d) = \min_{s' \in E^{TrulyReach} \cup E^{NotTrulyReach}} ea(s') + spp(s', d)$$

because the earliest appearance of any $s' \in E^{NotTrulyReach}$ diverges towards infinity. As a consequence, the earliest appearances of reachable nodes will eventually converge to their true values.

This result can be extended to actions with more than one 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.3 Proof that late nodes are unreachable (Proposition 4.2)

Proposition 4.2. *If a node is put in the late set, then it is not reachable.*

The proof is split into several definitions and proposition to facilitate reading. We use a graph formalism for the rest of this proof: a node is either a fluent or an elementary action. There is an edge from x to y if 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' start or from the action' start to the effect.

A.3.1 Self-dependent set

We first identify sufficient conditions to declare a set of nodes unreachable. A node X is the 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 node that would be updated every time its earliest appearance is modified.

Definition A.1 (Predecessor cycle)**.** A *predecessor cycle* is a sequence $A_1 \rightarrow a_1 \rightarrow A_2 \rightarrow a_2 \ldots A_n \rightarrow a_n \rightarrow A_1$ 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.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 predecessors 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, the cycle has a strictly positive length (otherwise it would not have resulted in a greater value). □

Proposition A.2. *In a predecessor cycle, at least one effect edge $A_i \rightarrow a_i$ 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 not 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 (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 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.3. *If a node n is part of a self-dependent set Ω, then n is not reachable.*

Proof. We first show that there is a predecessor cycle composed exclusively of nodes in Ω. All nodes in 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 in this cycle without altering the problem. This means that one fluent f in Ω is deprived of one of its achievers. 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 infeasible. □

Example A.1. The graph in Figure 14 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 $b \to A_1 \to a \to B \to b$ 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 predecessor cycle $b \to A_2 \to a \to B \to b$. 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.

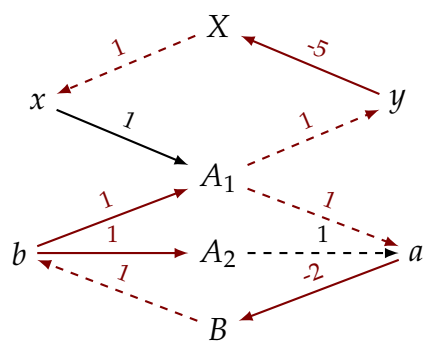


Figure 14: Problem in graph representation. Edges in red represent a possible assignment of predecessors at some point in the propagation.

A.3.2 From propagation to the identification of self-dependent set

We have identified sufficient conditions to declare a group of nodes unreachable. We now show how the identification of such a set can be integrated into 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.

$$\forall x \notin \mathcal{L}, y \in \mathcal{L}, ea(x) + d_{max} < ea(y) \tag{3}$$

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.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.

Let Y be the current predecessor of a node $X \in \mathcal{L}$ and e_{YX} the edge from Y to X. At the last update of X, $ea(X)$ was set to $ea(Y) + lbl(e_{YX})$. We know that $ea(Y)$ can only increase and that $lbl(e_{YX}) \leq d_{max}$. Consequently, we still have $ea(Y) + d_{max} \geq ea(X)$, meaning that Y is necessarily in $\mathcal{L}$. We have shown that if X is in $\mathcal{L}$, $pred(X)$ is in $\mathcal{L}$.

Next let x be 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 therefore in $\mathcal{L}$ as well. □

Corollary 1. *Any node in a late set is not reachable.*

Proposition A.5. *Any non-reachable node will eventually be part of a late set.*

Proof. All earliest appearances of reachable nodes converge towards a finite value. On the other hand, earliest appearances of non-reachable nodes diverge towards $+\infty$. At some point, the earliest appearance of unreachable nodes will be greater by d_{max} than the latest reachable node. □

A possible implementation for detecting unreachable nodes in Algorithm 2 is thus to check whether some nodes are identifiable as late during propagation and delete all those nodes. Such an implementation is guaranteed to finish and remove all unreachable nodes.

B Example of ANML domains

B.1 Dock Worker

Below is a partial view of the dock-worker domain that served as an illustration of the representation in this paper.

```
/*** Types, functions, state variables ***/
type Truck with {
  // loc: Time x Truck => Dock
  fluent Dock loc;
};
type Container with {
  fluent (Dock or Truck or Ship) pos;
};
...

// travel_time: Dock x Dock => Integer
constant int travel_time(Dock d1, Dock
    d2);
// connected: Dock x Dock => Boolean
constant bool connected(Dock d1, Dock d2);


/*** Actions ***/

// move action of Figure 1
action move(Truck r, Dock d1, Dock d2) {
  duration := travel_time(d1,d2);
  [all] r.loc == d1 :-> d2;
  [start,t] occupant(d1) == r :-> NIL;
  [t2,end] occupant(d2) == NIL :-> r;
  t < t2;
};

// the two high-level actions achieving the
    transport task (Figure 2).
action transport(Container c, Dock d) {
  motivated; // task-dependant
```

```
  // m1-transport
  :decomposition{
    [all] c.pos == d;
  };

  // m2-transport
  :decomposition{
    constant Truck r;
    constant Dock ds;
    connected(ds,d);
    ds != d;
    [start] r.loc == ds;
    [start] c.pos == ds;
    // three totally ordered subtasks
    [all] ordered(
      load(r,c,ds),
      move(r,ds,d),
      unload(r,c,d)
    );
  };
};


/*** Instances and Constants ***/
instance Dock dock1, dock2, dock3;
instance Truck r1, r2;
instance Ship ship1;

travel_time(dock1, dock2) := 7;
travel_time(dock2, dock3) := 9;


/*** Problem Statement (Figure 3) ***/

// initial state and expected evolution
[start] r1.loc := dock1;
[start] r2.loc := dock2;
[start] c1.pos := ship1;
[start+10] ship1.docked := pier1;
[t_undock] ship1.docked := NIL;
start+20 <= t_undock; t_undock <=
    start+30;

// goals and tasks
[end] r1.loc == dock1;
[end] r1.loc == dock2;
[start,end] contains transport(c1,dock3);
```

B.2 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; // i.e. task-dependent
  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; // i.e. task-dependent
  :decomposition {
    [all] on(a) == b;
  };
  :decomposition {
    [all] stack(a,b);
  };
};
```

B.3 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; // i.e. task-dependent
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == TABLE :-> HAND;
  [all] handempty == true :-> false;
};

action putdown(Block b) {
  motivated; // i.e. task-dependent
  duration := 5;
  [all] clear(b) == true;
  [all] on(b) == HAND :-> TABLE;
  [all] handempty == false :-> true;
};

action stack(Block b, Block c) {
  motivated; // i.e. task-dependent
  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; // i.e. task-dependent
  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; // i.e. task-dependent
  :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; // i.e. task-dependent
  :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);

  };
};
```