Noname manuscript No.
(will be inserted by the editor)

HTN-Like Solutions for Classical Planning Problems: An Application to BDI Agent Systems

Lavindra de Silva · Lin Padgham ·
Sebastian Sardina

Abstract In this paper we explore the question of what characterises a desirable plan of action and how such a plan could be computed, in the context of systems that already possess a certain amount of hierarchical domain knowledge. In contrast to past work in this setting, which focuses on generating low-level plans, losing much of the domain knowledge inherent in such systems, we argue that plans ought to be HTN-like or *abstract*, i.e., re-use and respect the user-supplied know-how in the underlying domain. In doing so, we recognise an intrinsic tension between striving for abstract plans but ensuring that unnecessary actions, not linked to the specific goal to be achieved, are avoided. We explore this tension by characterising the set of "ideal" abstract plans that are non-redundant but maximally abstract, and then develop a more limited yet feasible account in which a *given* (arbitrary) abstract plan is "specialised" into one such non-redundant plan that is as abstract as possible. We present an algorithm that can compute such specialisations, and analyse the theoretical properties of our proposal.

Keywords Abstract Solutions · Classical Planning · HTN Planning

1 Introduction

Hierarchical Task Network (HTN) planning [16, 19] is a well-understood and successful approach to planning, via the hierarchical and context-based decomposition of subgoals using supplied 'standard operating procedures'. This kind of reasoning is also adopted by Belief-Desire-Intention (BDI) agent systems [37], which are a popular approach to (soft) real-time reasoning and control in complex and dynamic environments [6, 25], such as Unmanned Autonomous Vehicles (UAVs) [46], Air Traffic Control [28], and robot supervision [1]. While HTN planners perform complete "lookahead" over subgoal decompositions

This work is based on and extends that in [42]. Most of this work was carried out while the first author was at RMIT and the University of Nottingham.

Lavindra de Silva
University of Cambridge, Cambridge, UK
E-mail: lavindra.desilva@eng.cam.ac.uk

Lin Padgham, Sebastian Sardina
RMIT University, Melbourne, Australia
E-mail: {lin.padgham,sebastian.sardina}@rmit.edu.au

in order to guarantee that they are achievable, BDI agents typically interleave this process with execution in the real world, lowering the likelihood of the reasoning becoming outdated due to environmental changes by the time execution happens. These agents can also quickly recover from failure that is caused by environmental changes, by choosing and pursuing a suitable alternative from a library of supplied recipes. While BDI systems have been extended to enable built-in HTN-style lookahead planning [39], it is sometimes also important to be able to plan from *first principles* [26]. Such reasoning is useful, or even mandatory when agents need to avoid undesirable outcomes, or devise novel recipes, not already supplied, to achieve their goals. In this paper, we use the HTN formalism, which has an established translation from typical BDI formalisms [39, 38], to characterise desirable classes of such recipes, and we explore how they could be computed.

Past work on classical (first principles) planning in BDI systems has focused on producing plans comprising only basic capabilities (actions) of the agent [13, 47, 33, 10, 31]. While conceptually simple, aiming for low-level, primitive plans (those with actions only) does not take into account the on-the-fly, hierarchical decompositional nature of the BDI approach. Indeed, primitive plans also do not reflect typical BDI recipes. When engaging in classical planning, BDI agents should instead strive for *hybrid* plans, i.e., plans built from both basic steps as well as *abstract* ones, where the latter is obtained from the agent's domain comprising hierarchical know-how information. Hybrid-plans are particularly appealing in the context of BDI systems. First, they tend to preserve so-called *"user-intent"* [27, 22], i.e., the principle that such domain knowledge describes the behaviour that the programmer deems acceptable for the agent to carry out.[1] BDI agents store such knowledge in the plan-library, comprising hierarchical, procedural structures to address the various goals relevant to the system, and which generally encodes non-functional requirements. Thus, hybrid-plans preserve user-intent by re-using and respecting the agent's existing domain knowledge. Secondly, hybrid-plans offer *flexibility* and *robustness*: if a refinement of an abstract step in a hybrid-plan happens to fail at run-time, another refinement could be tried. Finally, such plans promote *compactness*, which can be convenient when presenting and explaining the novel behaviour synthesised (to a human, for example).

The question that arises, then, is: *what characterises a preferred or an adequate hybrid-plan for a goal in the context of a BDI agent system, which already possesses a certain amount of know-how information?* In this paper, we address this question by developing an account of hybrid-plans based on the two fundamental principles of *abstraction* and *non-redundancy*. As described above, abstraction refers to the possibility of having plans containing abstract steps that ought to be refined, using knowledge encoded in the plan-library, into executable actions. Non-redundancy, in turn, states that no redundant steps, i.e., actions not needed to achieve the goal at hand, are used. The fact is that abstraction typically comes at the expense of increased redundancy, as more abstract steps are generally associated with a larger collection of actions. More importantly, even if the individual, manually built abstract steps do not yield redundant actions, this may not still be the case when the former are combined at runtime to form a novel hybrid-plan. By balancing abstraction and non-redundancy, our preferred hybrid-plans avoid redundant steps while promoting user-intent and knowledge re-use, as well as flexibility and robustness.

Example 1. Consider a Mars Rover exploring the surface of Mars. A part of our agent's domain is shown in Figure 1. The top-level task in the hierarchy is to explore a given soil location $L2$ from the current location $L1$. This is achieved by recipe (method) m_0, which prescribes navigating from $L1$ to $L2$ and then doing a soil experiment. Navigation uses m_1,

[1] Technically, any sequence of actions performed must be 'parseable' by a hierarchy in the domain [27].

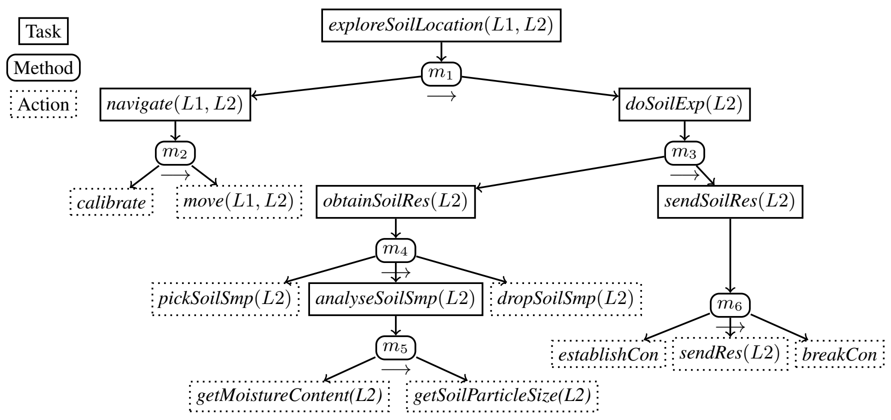


Fig. 1: A simple Mars Rover agent. An arrow below a recipe/method indicates that its steps are ordered from left to right.

which calibrates some instruments and moves from $L1$ to $L2$, and doing the experiment uses m_3, which obtains soil results from $L2$ and sends them to the lander. The former amounts to picking a soil sample from $L2$, analysing it, and discarding it, and the remaining tasks are refined into the actions shown (e.g. establishing a connection with the lander).

Suppose that a recipe $\hat{m}$ (not shown) in the larger domain fails on execution, and the agent performs planning to achieve the goal state where soil results have been sent for rock locations $r2$ and $r3$. Planning might yield the new recipe h in Figure 2a, by considering the tasks available in the entire domain, and reusing suitable ones. Now consider h's execution/decomposition trace (Figure 2c). Observe that breaking the connection after sending results for $r2$, and then re-establishing it before sending results for $r3$ are unnecessary, or redundant steps,[2] brought about by the inclusion of task *doSoilExp*. What we would prefer to have instead is the *non-redundant* yet *maximally* abstract hybrid-plan h' (Figure 2b). This avoids the redundancy produced by h, but respects much of its inherent user-intent—the structure of the abstract hierarchies supplied by the user. In particular, it retains *navigate* and *obtainSoilRes* and their order, allowing us to use BDI-style failure recovery and achieve these in an alternative manner to that shown here, if such existed and was warranted by the situation during execution. (As usual, if such 'local' recovery efforts fail, recovery efforts for $\hat{m}$ will continue.) The replacement of each of *doSoilExp* and *sendSoilRes* with a subset of their components is clearly motivated in order to remove redundancy. ■

The roadmap and contributions of this paper are as follows. In Section 2, we provide the background material with a focus on HTNs, which provide a fitting planning formalism that is closely related to BDI systems in both syntax and semantics [39, 38]. While the HTN planning semantics is not necessary for computing an *arbitrary* hybrid-solution, which we define in Section 3, the semantics is needed in Section 4, where we characterise the set of "ideal" hybrid-plans, which are non-redundant as well as maximally compact and abstract. The weaker but more feasible notion of a "preferred specialisation", where a given (arbitrary) hybrid-solution is "specialised" into one that is non-redundant but preserves abstraction as much as possible, is developed in Section 5. In Sections 4 and 5, we also show

[2] As with TCP/IP, we assume that keeping a connection open consumes negligible battery power.

1. *navigate*$(r1, r2)$
2. *doSoilExp*$(r2)$
3. *navigate*$(r2, r3)$
4. *doSoilExp*$(r3)$

(a) Hybrid-solution h

1. *navigate*$(r1, r2)$
2. *obtainSoilRes*$(r2)$
3. *establishCon*
4. *sendRes*$(r2)$
5. *navigate*$(r2, r3)$
6. *obtainSoilRes*$(r3)$
7. *sendRes*$(r3)$
8. *breakCon*

(b) Hybrid-solution h'

1. *navigate*$(r1, r2)$
   (A) *calibrate*
   (B) *move*$(r1, r2)$
2. *doSoilExp*$(r2)$
   (A) *obtainSoilRes*$(r2)$
      (i) *pickSoilSmp*$(r2)$
      (ii) *analyseSoilSmp*$(r2)$
         (a) *getMContent*$(r2)$
         (b) *getSPSize*$(r2)$
      (iii) *dropSoilSmp*$(r2)$
   (B) *sendSoilRes*$(r2)$
      (i) *establishCon*
      (ii) *sendRes*$(r2)$
      (iii) ***breakCon***

3. *navigate*$(r2, r3)$
   (A) *calibrate*
   (B) *move*$(r2, r3)$
4. *doSoilExp*$(r3)$
   (A) *obtainSoilRes*$(r3)$
      (i) *pickSoilSmp*$(r3)$
      (ii) *analyseSoilSmp*$(r3)$
         (a) *getMContent*$(r3)$
         (b) *getSPSize*$(r3)$
      (iii) *dropSoilSmp*$(r3)$
   (B) *sendSoilRes*$(r3)$
      (i) ***establishCon***
      (ii) *sendRes*$(r3)$
      (iii) *breakCon*

(c) Execution trace of hybrid-solution h (shown as two halves)

Fig. 2: (a) A redundant hybrid-solution h; (b) a hybrid-solution h' with redundancy (actions in bold) removed; and (c) the execution trace of h (with some actions acronymed).

that ideal hybrid-plans and preferred specialisations always exist, provided the planning problem itself admits a solution. More importantly, we prove that any ideal hybrid-plan occurring in a decomposition trace is indeed also a preferred specialisation of the trace, thus providing support for the latter concept. In Section 6 we give an algorithm for obtaining such specialisations, and in Sections 7 and 8 we discuss related work and draw conclusions.

2 Background

In this section, we briefly review classical planning, HTN planning as described in [16, 19], and BDI agent systems. Our definitions use Prolog conventions: predicate symbols and constants begin with lower case, and variables with upper case.

2.1 Classical Planning

Classical Planning deals with the synthesis, from first principles, of a sequence of actions (operators) that achieves a given goal when executed from the initial state. We follow the STRIPS-based account of planning [20] as presented in [26].

A *classical planning problem* is a tuple $\mathcal{C} = \langle \mathcal{I}, \mathcal{G}, Op \rangle$, where $\mathcal{I}$ is the initial state, $\mathcal{G}$ the *goal condition*, and *Op* the set of available operators. Roughly speaking, a solution to a planning problem is a sequence of operators that, when executed, transform the initial state into a goal state. Let us make this more precise. A *state* is a set of ground atoms representing the propositions that hold true in the world. A *goal condition* is a conjunction of literals, which must be true in any goal state. An *operator* is a tuple $op = \langle name(op), pre(op), add(op), del(op) \rangle$, where *(i)* $name(op) = act(\mathbf{x})$, the operator's name, is a symbol followed by a vector $\mathbf{x}$ of distinct variables such that all free variables in $pre(op), add(op)$, and $del(op)$ also occur in $\mathbf{x}$; *(ii)* $pre(op)$ is a set of literals representing the operator's precondition; and *(iii)* $add(op)$ and $del(op)$ are the *add-* and *delete-list*, respectively, which are sets of atoms representing the operator's effects. An *action* is a ground instance of the operator's name, and a *primitive plan* σ is a sequence of actions.

Let us formally define what it means for a primitive plan to be a solution for a planning problem. The *result* of applying (i.e., executing) an action *act* in a state $\mathcal{S}$ relative to operator set *Op*, denoted $\mathit{Res}(\mathit{act}, \mathcal{S}, \mathit{Op})$, is defined as follows (recall *act* is ground):

$$\mathit{Res}(\mathit{act}, \mathcal{S}, \mathit{Op}) = \begin{cases} \big(\mathcal{S} \setminus \mathit{del}(\mathit{op})\theta\big) \cup \mathit{add}(\mathit{op})\theta & \text{if } \mathit{op} \in \mathit{Op}, \mathit{act} = \mathit{name}(\mathit{op})\theta \\ & \quad \text{and } \mathcal{S} \models \mathit{pre}(\mathit{op})\theta; \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

Symbol θ denotes a *substitution* [29], i.e., a finite set of elements $\{x_1/t_1, \ldots, x_n/t_n\}$ in which $x_1, \ldots, x_n$ are distinct variables and each t_i is a term such that $t_i \neq x_i$. As usual we use $E\theta$ to denote the expression obtained from any expression E by simultaneously replacing each occurrence of x_i in E with t_i, for all $i \in \{1, \ldots, n\}$.

Next, we state what it means to apply a sequence of actions. The *result* of applying (executing) a sequence of actions $\sigma = a_1 \cdot \ldots \cdot a_n$ from a state $\mathcal{S}$ relative to operator set *Op*, denoted $\mathit{Res}^*(a_1 \cdot \ldots \cdot a_n, \mathcal{S}, \mathit{Op})$, is defined inductively as follows:

$$\mathit{Res}^*(a_1 \cdot \ldots \cdot a_n, \mathcal{S}, \mathit{Op}) = \begin{cases} \mathit{Res}(a_1, \mathcal{S}, \mathit{Op}) & \text{if } n = 1; \\ \mathit{Res}^*(a_2 \cdot \ldots \cdot a_n, \mathit{Res}(a_1, \mathcal{S}, \mathit{Op}), \mathit{Op}) & \text{if } n > 1; \\ \mathcal{S} & \text{otherwise.} \end{cases}$$

Intuitively, Res^* yields the resulting state after executing the whole sequence of actions from state $\mathcal{S}$. Finally, a primitive plan σ is a *primitive solution* for a classical planning problem $\mathcal{C} = \langle \mathcal{I}, \mathcal{G}, \mathit{Op} \rangle$ iff $\mathit{Res}^*(\sigma, \mathcal{I}, \mathit{Op}) \models \mathcal{G}$, i.e., the resulting state on executing σ from state $\mathcal{I}$ satisfies the goal condition $\mathcal{G}$.

2.2 BDI Agents and HTN Planning

While classical planners focus on bringing about states of affairs (i.e., "goals-to-be") from first principles, BDI agent systems and HTN planners aim at executing or solving *abstract/compound tasks* (i.e., "goals-to-do"). In a nutshell, both the BDI execution engine and the HTN planning process involve decomposing *abstract* tasks repeatedly into less abstract ones, up to executable *primitive* tasks, by relying on user-supplied know-how information. Whereas HTN systems are concerned with hypothetical *off-line reasoning* about actions and their potential interactions within a pursued plan for solving a task, BDI agent systems focus on the effective *online execution* of hierarchical agent programs in complex and dynamic environments. There are many BDI-style agent programming languages and systems available (e.g., [7, 11, 38]) as well as HTN planning systems (e.g., [35, 36, 40, 43, 15]).

Despite their different purposes, however, BDI systems and HTN planners share many similarities [14, 12, 39, 38]. In particular, and crucially for our work, both approaches make use of procedural domain control knowledge, encoded in the form of reduction methods/recipes. These aim at capturing the standard operational procedures of the domain of concern, which allow more focused reasoning as well as the specification of non-functional requirements. Roughly speaking, reduction methods/recipes are of the form $e : \psi \leftarrow P$, encoding that program P is a "reasonable strategy" to resolve abstract task/goal e when precondition ψ holds true. (Importantly, program P may contain primitive, directly executable steps as well as further abstract tasks/goals.) A precondition is similar to that in the SHOP [35] and SHOP2 [36] HTN planning systems, which can be expressed as constraints in the HTN formalism that we use in this paper. The objective of both the HTN and BDI approach is to reduce higher-level HTN tasks or BDI goals into lower-level ones, by referring to a repository of recipes (called a *method library* in HTN systems and a *plan library* in BDI systems), until primitive, executable actions are reached.

Since it was proved already that the underlying semantics of BDI execution and the HTN planning process coincide [39, 38], we shall follow, from now on, the syntax and semantics of HTN planning as described in [16, 19]. An HTN *planning problem* $\mathcal{P}$ is a tuple $\langle d, \mathcal{I}, \mathcal{D} \rangle$, where d is a *task network* and $\mathcal{I}$ is an initial state. Element $\mathcal{D}$ is an HTN *planning domain*, which is a tuple $\langle Op, Me \rangle$, where *Op* is a set of HTN operators and *Me* is a set of methods. The objective of the HTN planner is to solve task network d, by starting from state $\mathcal{I}$, and using the sets *Me* and *Op*. A task network is a partially ordered set of tasks, where a task is either *compound* or *primitive*, both of which are of the form $t(\mathbf{o})$, where $\mathbf{o}$ is a vector of function-free terms (i.e., a term is either a variable or constant). Thus, an action is a ground primitive task. Formally, a *task network* is a syntactic construct of the form:

$$[\![\{(n_1 : t_1), \ldots, (n_m : t_m)\}, \phi]\!],$$

where the first component is a set of labelled tasks, and the second is a *task network formula* having constraints that must be respected; each n_i is a label used to uniquely identify a task occurring in the network, where $n_i \in \mathbb{N}_0$.

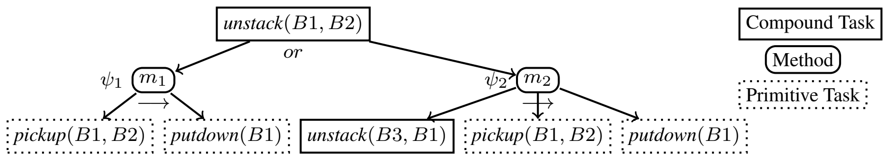
A *task network formula* is a Boolean formula constructed from negation, conjunction, disjunction, and the following: *(i) variable binding constraints* of the form $(o = o')$, where o and o' are variables or constants; *(ii) ordering constraints* of the form $(n \prec n')$, sometimes with parenthesis omitted, where n and n' are task labels; and *state constraints* of the form (l, n), (n, l) and (n, l, n'), where l is a literal and n and n' are task labels. A variable binding constraint $(o = o')$ indicates that o must be equal to o'. An ordering constraint $(n \prec n')$ indicates that the task with label n should precede the one with label n'; thus, a task network is said to be *totally ordered* if its task network formula entails an ordering constraint for every pair of unique labels occurring in the task network. State constraints (l, n) (resp. (n, l)) indicate that literal l should hold immediately before (resp. after) the task with label n. Finally, a state constraint (n, l, n') indicates that literal l should hold between the tasks with labels n and n'. Task labels can also be of the form *first*$[n_1, \ldots, n_m]$ and *last*$[n_1, \ldots, n_m]$, which allow referring, respectively, to the task that starts first, and to the task that ends last among the set $\{n_1, \ldots, n_m\}$.

Next, we define the structures used to realise the tasks occurring in a task network. If the task is primitive, it is realised via an operator as defined above, and there is exactly one operator in *Op* with a name that unifies with act. If instead the task is compound, it is achieved via a method. A *method* is of the form (t, d), where t is a compound task and d is a task network. A compound task may have more than one associated method in *Me*.

Example 2. We illustrate some of the above constructs with the Blocks World HTN domain $\mathcal{D} = \langle Op, Me = \{m_1, m_2\} \rangle$ depicted in Figure 3. The top-level compound task in this domain is *unstack*$(B1, B2)$, which is used to move a block $B1$ that is on top of a block $B2$ onto the table. Observe that method $m_1 = (\textit{unstack}(B1, B2), d')$, where task network d' is

$$[\![\{(n_1 : \textit{pickup}(B1, B2)), (n_2 : \textit{putdown}(B1))\}, (n_1 \prec n_2) \wedge \phi]\!],$$

with $\phi = (\textit{clear}(B1), n_1) \wedge (\textit{on}(B1, B2), n_1) \wedge (\textit{armEmpty}, n_1)$. The primitive task $pickup(B1, B2)$ and *putdown*$(B1)$ respectively picks up a block $B1$ that is on top of a block $B2$, and places a block $B1$—that is being held—on the table. The task network formula of d' requires *pickup*$(B1, B2)$ to precede *putdown*$(B1)$, and that initially (i.e., before picking up), $B1$ is clear, $B1$ is on top of $B2$, and the robot's arm is empty.



$\psi_1 = \mathit{clear}(B1) \wedge \mathit{on}(B1, B2) \wedge \mathit{armEmpty}$ $\quad\psi_2 = \mathit{on}(B1, B2) \wedge \mathit{on}(B3, B1) \wedge \mathit{armEmpty}$

Primitive Task	Precondition	Effects
$\mathit{putdown}(B1)$	$\mathit{holding}(B1)$	$\neg\mathit{holding}(B1) \wedge \mathit{armEmpty} \wedge \mathit{on}(B1, T)$
$\mathit{pickup}(B1, B2)$	$\mathit{on}(B1, B2) \wedge \mathit{clear}(B1) \wedge \mathit{armEmpty}$	$\mathit{holding}(B1) \wedge \neg\mathit{armEmpty} \wedge \neg\mathit{on}(B1, B2) \wedge \mathit{clear}(B2)$

Fig. 3: A simplified depiction of an HTN domain. An arrow below a method indicates that its tasks are ordered from left to right, and ψ_1, ψ_2 are "preconditions" of respectively m_1, m_2.

Observe from the figure that method $m_2 = (\mathit{unstack}(B1, B2), d'')$, where task network $d'' = [\![s, \phi_1 \wedge \phi_2]\!]$ with

$$\begin{aligned} s &= \{(n_1 : \mathit{unstack}(B3, B1)), (n_2 : \mathit{pickup}(B1, B2)), (n_3 : \mathit{putdown}(B1))\}, \\ \phi_1 &= (n_1 \prec n_2) \wedge (n_2 \prec n_3), \text{ and} \\ \phi_2 &= (\mathit{on}(B1, B2), n_1) \wedge (\mathit{on}(B3, B1), n_1) \wedge (\mathit{armEmpty}, n_1). \end{aligned}$$

Thus, while m_1 moves a block $B1$ only if it has no other blocks on top of it, m_2 handles the case where there are such blocks, by first clearing $B1$—i.e., recursively moving each block on top of it onto the table—and then moving $B1$ onto the table. ■

Given an HTN planning problem $\mathcal{P} = \langle d, \mathcal{I}, \mathcal{D}\rangle$, with $\mathcal{D} = \langle \mathit{Op}, \mathit{Me}\rangle$, the HTN planning process works as follows. First, an applicable reduction method (i.e., one whose precondition is met in the current state) is selected from *Me* and applied to some compound task in d. This will result in a new, and typically "more primitive" task network d'. Then, another reduction method is applied to some task in d', and this process continues until a task network is obtained that contains only primitive tasks. At any stage during the planning process, if no applicable method can be found for a compound task, the planner "backtracks" and tries an alternative reduction for a compound task previously reduced.

To be more precise about the above process, we first define a reduction (decomposition). Let $d = [\![s, \phi]\!]$ be a task network, $(n : t) \in s$ a (labelled) compound task occurring in d, and $m = (t', d') \in \mathit{Me}$ a method that is relevant for t (i.e., t and t' unify). Then, $\mathit{reduce}(d, n, m)$ denotes the task network resulting from decomposing task $(n : t)$ in task network d via method m. Informally, decomposition involves updating both the set s in d, by replacing $(n : t)$ with the tasks in d' (by arbitrarily renaming task labels), and the constraints ϕ in d to take into account those in d'. The set of all reductions of a task network d is denoted $\mathit{red}(d, \mathcal{D})$. The formal definitions of reductions can be found in Appendix A (Definition 15).

If all compound tasks occurring in a given initial task network have been replaced by primitive tasks via reductions, the last step is to find a *completion* of the resulting task network, i.e., an ordering and grounding of its primitive tasks that conforms with the constraints imposed on those tasks by the network. More precisely, a plan σ is a completion of a primitive task network d at state $\mathcal{I}$, denoted $\sigma \in \mathit{comp}(d, \mathcal{I}, \mathcal{D})$, if σ is a total ordering of the primitive tasks in a ground instance of d such that σ is executable in $\mathcal{I}$ (i.e., all preconditions associated with primitive tasks in σ are satisfied), and σ satisfies the constraint formula in d. The formal definition of a completion can be found in Appendix A (Definition 14).

By taking the sets $\mathit{red}(d,\mathcal{D})$ and $\mathit{comp}(d,\mathcal{I},\mathcal{D})$ we can now define the set of plans $\mathit{sol}(d,\mathcal{I},\mathcal{D})$ that solves an HTN planning problem $\mathcal{P} = \langle d,\mathcal{I},\mathcal{D}\rangle$ as follows: $\mathit{sol}(d,\mathcal{I},\mathcal{D}) = \bigcup_{n\in\mathbb{N}_0} \mathit{sol}^n(d,\mathcal{I},\mathcal{D})$, where $\mathit{sol}^n(d,\mathcal{I},\mathcal{D})$ is defined inductively as

$$\begin{aligned} \mathit{sol}^0(d,\mathcal{I},\mathcal{D}) &= \mathit{comp}(d,\mathcal{I},\mathcal{D}),\\ \mathit{sol}^{n+1}(d,\mathcal{I},\mathcal{D}) &= \mathit{sol}^n(d,\mathcal{I},\mathcal{D}) \cup \bigcup_{d'\in \mathit{red}(d,\mathcal{D})} \mathit{sol}^n(d',\mathcal{I},\mathcal{D}). \end{aligned}$$

Intuitively, the set of primitive plans that solves an HTN planning problem $\langle d,\mathcal{I},\mathcal{D}\rangle$ is the set of all completions of all primitive task networks that can be obtained from zero or more reductions of d. We call such plans *primitive plan solutions* to distinguish them from primitive solutions which achieve some goal state, and from primitive plans which are arbitrary sequences of actions. We also use the notion of a <u>*labelled*</u> primitive plan, which is a possibly empty sequence $\tau = (n_1 : t_1) \cdot \ldots \cdot (n_m : t_m)$ of labelled tasks. We sometimes treat a labelled primitive plan τ as a (unlabelled) primitive plan σ with the obvious meaning.

Example 3. We describe reductions and completions with the HTN domain $\mathcal{D}$ in Figure 3, and the task network $d_1 = [\![s_1,\phi_1]\!]$, where $s_1 = \{(n : \mathit{unstack}(b1,b2))\}$ and $\phi_1 = \mathit{true}$. Suppose we have the HTN planning problem $\mathcal{P} = \langle d_1,\mathcal{I},\mathcal{D}\rangle$, where $\mathcal{I}$ contains the following facts: $\mathit{on}(b2,\mathit{table})$, $\mathit{on}(b1,b2)$, $\mathit{on}(b3,b1)$, $\mathit{clear}(b3)$, and $\mathit{armEmpty}$. Then, observe that the reduction of the labelled compound task $(n : \mathit{unstack}(b1,b2)) \in s_1$ via method m_1, or $\mathit{reduce}(d_1,n,m_1)$, results in the primitive task network

$$d_2 = [\![\{(n_1 : \mathit{pickup}(b1,b2)),(n_2 : \mathit{putdown}(b1))\},(n_1 \prec n_2) \wedge \phi_2]\!],$$

where $\phi_2 = (\mathit{clear}(b1),n_1) \wedge (\mathit{on}(b1,b2),n_1) \wedge (\mathit{armEmpty},n_1)$. However, the completion of d_2 is $\mathit{comp}(d_2,\mathcal{I},\mathcal{D}) = \emptyset$, as $(\mathit{clear}(b1),n_1)$ does not hold in state $\mathcal{I}$ (literal $\mathit{clear}(b1)$ is not true in $\mathcal{I}$). Consider, instead, the reduction of labelled task $(n : \mathit{unstack}(b1,b2)) \in s_1$ via method m_2. The result of this decomposition is the following task network

$$d_3 = [\![\{(n_1 : \mathit{unstack}(b3,b1)),(n_2 : \mathit{pickup}(b1,b2)),(n_3 : \mathit{putdown}(b1))\},\phi_3]\!],$$

where formula $\phi_3 = (n_1 \prec n_2) \wedge (n_2 \prec n_3) \wedge (\mathit{on}(b1,b2),n_1) \wedge (\mathit{on}(b3,b1),n_1) \wedge (\mathit{armEmpty},n_1)$. Since there is a compound task *unstack* occurring in d_3, it needs to be reduced further before a primitive task network can be obtained. Suppose that method m_1 is used for this reduction. Then, the resulting primitive task network is $d_4 = [\![s_4,\phi_4]\!]$ where

$$\begin{aligned} s_4 &= \{(n_4 : \mathit{pickup}(b3,b1)),(n_5 : \mathit{putdown}(b3)),\\ &\qquad (n_2 : \mathit{pickup}(b1,b2)),(n_3 : \mathit{putdown}(b1))\},\\ \phi_4 &= (n_4 \prec n_2) \wedge (n_5 \prec n_2) \wedge (n_2 \prec n_3) \wedge (n_4 \prec n_5) \wedge \phi_5, \text{ and}\\ \phi_5 &= (\mathit{clear}(b3),n_4) \wedge (\mathit{on}(b3,b1),n_4) \wedge (\mathit{armEmpty},n_4) \wedge (\mathit{on}(b1,b2),n_4). \end{aligned}$$

Observe that the contents of m_1's task network have been included in d_4. That is, *(i)* m_1's constraint formula has been added as a conjunction to ϕ_4; *(ii)* labelled tasks in m_1 have been added, after renaming task labels, to s_4; and *(iii)* the constraints in d_4—e.g. $(n_1 \prec n_2)$—have been updated to accommodate the new task labels. The final step is to obtain the completion $\mathit{comp}(d_4,\mathcal{I},\mathcal{D})$ of task network d_4, which yields the primitive plan solution $\mathit{pickup}(b3,b1)\cdot\mathit{putdown}(b3)\cdot\mathit{pickup}(b1,b2)\cdot\mathit{putdown}(b1)$. This is a primitive plan solution because it is executable from initial state $\mathcal{I}$ and ϕ_5 is satisfied with respect to $\mathcal{I}$. ■

2.3 Epsilon Tasks and Assumptions

In this paper we make use of "dummy" primitive tasks, which we call ϵ tasks. While ϵ tasks are still primitive tasks, they do not have a precondition and effect, and thereby amount to "doing nothing." Such tasks allow the inclusion of conditions within a method even when the method has no tasks occurring in it, and likewise, the specification of conditions that involve compound tasks even if the tasks eventually reduce into the empty set.

Example 4. To see why ϵ tasks are useful in HTN planning, consider an elevator domain having the following two methods for handling the compound task *go-to-bottom* (gtb), which keeps moving down one floor until the ground floor (floor 0) is reached:

$$(gtb, [\![\{(1 : \textit{move-down}), (2 : gtb)\}, (1 \prec 2) \wedge (\neg \textit{floor}(0), 1)]\!])$$
$$(gtb, [\![\{(1 : \epsilon)\}, (\textit{floor}(0), 1)]\!]).$$

Observe that without the ϵ task in the second method, there is no obvious way to encode that the elevator must stop moving down once the ground floor is reached.[3] ■

This paper makes the following assumptions. First, since ϵ tasks are used solely for specifying conditions on tasks as described above, we assume without loss of generality that any ϵ tasks occurring in primitive plans in the set $sol(d, \mathcal{I}, \mathcal{D})$ of HTN solutions have been removed. Our second, related, assumption is that an HTN method's associated set of labelled tasks is never empty: if the set does not mention any standard (non-ϵ) tasks, then it must mention the ϵ task. Finally, we assume that a task network's constraint formula does not mention any labels that do not also occur in the task network's set of labelled tasks.

3 Hybrid-Plans

The notion of a *hybrid-plan* forms the basis for many of the more involved notions that we introduce later. Intuitively, a hybrid-plan is a partially ordered set of steps, i.e., steps in the plan can be interleaved. Technically, a hybrid-plan is a *partially-ordered* plan [34], which, unlike a traditional one, can also contain (or can consist of) compound tasks.

Definition 1 (Hybrid-Plan). A *hybrid-plan* is an HTN task network $h = [\![s, \phi]\!]$, such that ϕ does not mention any state or variable binding constraints (i.e., ϕ is simply a conjunction of HTN ordering constraints).[4] ■

In this paper we investigate what we refer to as *hybrid planning*, which deals with synthesising hybrid-plans that can bring about a certain state of affairs (as in classical planning) by making use of the available domain knowledge (as in HTN planning). Hybrid planning finds solutions for *hybrid planning problems*. Formally, a *hybrid planning problem* is a tuple $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$, where $\mathcal{I}$ is an initial state, $\mathcal{G}$ is a goal condition, and $\mathcal{D}$ is an HTN domain.

[3] One may wonder whether the following encoding also works:
$(gtb, [\![\{(1 : \textit{move-down}), (2 : gtb)\}, (1 \prec 2) \wedge (\neg \textit{floor}(1), 1) \wedge (\neg \textit{floor}(0), 1)]\!])$
$(gtb, [\![\{(1 : \textit{move-down})\}, (\textit{floor}(1), 1)]\!])$.
These methods will not work when the elevator is at floor 0 in the initial state—because then neither of the methods will be applicable.

[4] Since state and variable binding constraints represent conditions that need to be achieved via planning, they can occur in a task network (e.g. one that is part of a planning problem) but not in a plan/solution.

Hybrid-plans that solve hybrid planning problems are called *hybrid-solutions*. More specifically, a hybrid-solution is a hybrid-plan that can be decomposed using the given domain knowledge into a primitive plan that brings about the goal condition. In what follows, any hybrid planning problem $\mathcal{H}$ is of the form $\langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$, and any planning domain $\mathcal{D}$ is of the form $\langle Op, Me \rangle$.

Definition 2 (Hybrid-Solution). Let h be a hybrid-plan, $\mathcal{H}$ a hybrid planning problem, and $\Sigma_h \subseteq sol(h, \mathcal{I}, \mathcal{D})$ a finite subset of primitive plan solutions. Then, h is a *hybrid-solution* for $\mathcal{H}$ *relative to* Σ_h iff $\Sigma_h \cap sol(\mathcal{I}, \mathcal{G}, Op) \neq \emptyset$, that is, if there is an HTN solution—a primitive plan—for h that achieves the goal.[5] ■

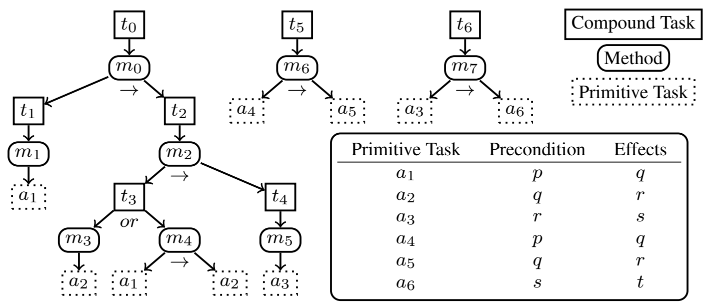
We are not concerned in this paper about *how* a hybrid-solution is obtained for the planning problem at hand: any approach (e.g. [44, 30]) may be used provided it yields a (valid) hybrid-solution as defined above. Note that our notion of hybrid planning is closer to classical planning than general HTN planning, which is undecidable [24, 19]. We outline the class of a hybrid planning problem with respect to the following two decision problems. First, given a ground hybrid planning problem $\mathcal{H}$, a hybrid-plan h, and a finite set of primitive plan solutions $\Sigma_h \subseteq sol(h, \mathcal{I}, \mathcal{D})$, the decision problem of whether h is a hybrid-solution for $\mathcal{H}$ relative to Σ_h is in **P**. This is because we simply need to check whether there exists a primitive solution $\sigma \in \Sigma_h$ for the classical planning problem $\langle \mathcal{I}, \mathcal{G}, Op \rangle$, which requires testing at most $|\Sigma_h|$ primitive plans (which we *know* are HTN solutions for h). Second, the decision problem of whether there exists a hybrid-plan h that is a hybrid-solution for $\mathcal{H}$ relative to some finite set $\Sigma_h \subseteq sol(h, \mathcal{I}, \mathcal{D})$ is in **PSPACE**. This is because deciding whether there exists a primitive solution $\sigma \in sol(\mathcal{I}, \mathcal{G}, Op)$ is in **PSPACE** [18], and any such solution can be straightforwardly represented as a (totally-ordered) hybrid-plan, which is also a hybrid-solution for $\mathcal{H}$ relative to $\{\sigma\}$.

Our first step, in the next section, is to develop the notion of an "ideal" hybrid-plan (Definition 7), as one that is non-redundant but at the same time maximally abstract and "minimal." Minimality ensures that it is not possible to make the plan any more compact by removing steps. Since it is unclear how such ideal plans can be computed within a reasonable amount of time, we define, in Section 5, a weaker, but computationally feasible notion of a "preferred specialisation" (Definition 12), which is an extraction of a desirable hybrid-plan from a particular decomposition trace produced by another hybrid-plan. We shall show properties of these notions that are in accordance with their intended meaning.

4 Ideal Hybrid-Plans

In order to develop an unambiguous definition of what makes a hybrid-plan "ideal", this section introduces the three inter-related notions *maximal-abstractness*, *minimality*, and *non-redundancy*. A *non-redundant* hybrid-solution for a hybrid planning problem is a hybrid-plan that yields (via HTN decompositions) a non-redundant primitive solution for the problem. A *minimal* hybrid-solution for a problem is one that is non-redundant, but no longer has this property as soon as one or more (primitive or compound) tasks are removed from the solution. Finally, a *maximally-abstract* hybrid-plan is one that does not contain any "subset" of compound tasks that could potentially be combined into a single (more) compound task.

[5] One could also imagine a stronger notion of a hybrid-solution where *all* HTN solutions for hybrid-plan h achieve the goal; in this paper, however, we only use the notion defined.



Primitive Task	Precondition	Effects
a_1	p	q
a_2	q	r
a_3	r	s
a_4	p	q
a_5	q	r
a_6	s	t

Fig. 4: A simple totally-ordered HTN domain. The table shows the preconditions and effects of the primitive tasks.

One concept that is intrinsic in these notions is that of a "refinement" of a hybrid-plan. In particular, a maximally-abstract hybrid-plan can be defined as one that is not the refinement of any other hybrid-plan. Intuitively, the refinements of a hybrid-plan (or task network) are all the "intermediate" or "partially reduced" task networks that may be encountered while searching for a primitive plan solution for the given hybrid-plan. The following example illustrates this notion of a refinement, as well as the notions of maximal-abstractness, non-redundancy, and minimality.

Example 5. Consider the HTN domain in Figure 4. If we take hybrid-solution t_0 (having just the one task), then its set of refinements with respect to the domain in the figure is basically the following set of hybrid-plans:[6]

$$\begin{array}{lllll}
\{t_0, & t_1 \cdot t_2, & t_1 \cdot t_3 \cdot t_4, & t_1 \cdot t_3 \cdot a_3, & t_1 \cdot a_2 \cdot t_4, \\
t_1 \cdot a_1 \cdot a_2 \cdot t_4, & t_1 \cdot a_2 \cdot a_3, & t_1 \cdot a_1 \cdot a_2 \cdot a_3, & a_1 \cdot t_2, & a_1 \cdot t_3 \cdot t_4, \\
a_1 \cdot t_3 \cdot a_3, & a_1 \cdot a_2 \cdot t_4, & a_1 \cdot a_1 \cdot a_2 \cdot t_4, & a_1 \cdot a_2 \cdot a_3, & a_1 \cdot a_1 \cdot a_2 \cdot a_3\}.
\end{array}$$

Refinement $t_1 \cdot a_1 \cdot a_2 \cdot t_4$ in the set is obtained by performing three reductions: first, t_0 is reduced via method m_0 to obtain task network $t_1 \cdot t_2$; second, $t_1 \cdot t_2$ is reduced via method m_2 to obtain task network $t_1 \cdot t_3 \cdot t_4$; and finally, $t_1 \cdot t_3 \cdot t_4$ is reduced via method m_4 to obtain task network $t_1 \cdot a_1 \cdot a_2 \cdot t_4$. The rest of the refinements are derived similarly.

Next, consider the table below, which shows some of the hybrid-solutions for the hybrid planning problem with initial state $\{p\}$, goal condition s, and the HTN domain in Figure 4.

HYBRID-SOL.	NON-REDUNDANT	MINIMAL	MAXIMALLY-ABSTRACT	IDEAL
t_0	√	√	√	√
t_2	√	√	√	√
$t_5 \cdot t_4$	√	√	√	√
$t_3 \cdot t_4$	√	√	×	×
$t_5 \cdot t_2$	×	×	√	×
$t_1 \cdot t_2$	√	×	×	×
$t_1 \cdot t_3 \cdot t_4$	√	×	×	×
$t_5 \cdot t_3 \cdot t_4$	×	×	×	×

Maximal-abstractness. From the above set of refinements we can see that hybrid-solution t_2 is maximally-abstract because it is not the refinement of any other hybrid-

[6] We use sequences here to represent totally ordered task networks having neither state nor variable binding constraints.

solution (t_0 does not have a refinement that matches t_2 alone). On the other hand, hybrid-solution $t_1 \cdot t_2$ is not maximally-abstract because it is a refinement of hybrid-solution t_0.

Redundancy. Hybrid-solution t_0 is non-redundant because it can produce the non-redundant primitive solution $a_1 \cdot a_2 \cdot a_3$ by selecting methods in the following order: m_0, m_1, m_2, m_3, and m_5. On the other hand, hybrid-solution $t_5 \cdot t_2$ is redundant because all of its primitive solutions (i.e., $a_4 \cdot a_5 \cdot a_2 \cdot a_3$ and $a_4 \cdot a_5 \cdot a_1 \cdot a_2 \cdot a_3$) are redundant. Finally, solution $a_4 \cdot a_5 \cdot a_2 \cdot a_3$ is redundant because it will remain a solution even if primitive tasks a_5 or a_2 are removed from it.

Minimality. Hybrid-solution $t_5 \cdot t_4$ is minimal because it is non-redundant and none of its proper subsequences (i.e., t_5 and t_4) are also non-redundant hybrid-solutions.[7] On the other hand, although $t_1 \cdot t_2$ is non-redundant, it is not minimal, because a proper subsequence of it—t_2—is also a non-redundant hybrid-solution. ■

4.1 Non-Redundant and Minimal Hybrid-Plans

We shall now be more precise about the notions described, starting with the notions of non-redundancy and minimality. To define non-redundancy, we extend the notion of a *perfect justification*, defined for primitive solutions in [21].

Definition 3 (Perfect Justification [21]). A primitive solution σ for a classical planning problem $\mathcal{C} = \langle \mathcal{I}, \mathcal{G}, Op \rangle$ is a *perfect justification* for $\mathcal{C}$ if there does not exist a proper subsequence σ' of σ such that σ' is a primitive solution for $\mathcal{C}$. ■

Thus, any given primitive solution that is not already a perfect justification can definitely yield one or more of them; given σ and $\mathcal{C}$ as above, the computational complexity of yielding a perfect justification for $\mathcal{C}$ from σ is NP-hard, and so is checking whether σ is a perfect justification for $\mathcal{C}$ [21]. Using the above definition we define a hybrid-solution as *non-redundant* if it can produce at least one perfect justification.

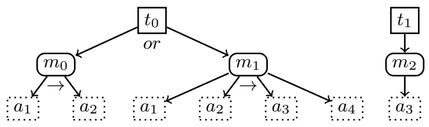
Definition 4 (Non-Redundant Hybrid-Solution). Let $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ be a hybrid planning problem, and h a hybrid-solution for $\mathcal{H}$ relative to a (finite) set of primitive plan solutions Σ_h. Then, h is a *non-redundant* hybrid-solution for $\mathcal{H}$ relative to Σ_h if there exists a $\sigma \in \Sigma_h$ such that σ is a perfect justification for problem $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. ■

Now we can define *minimality*. Intuitively, a non-redundant hybrid-solution h is minimal if there is no substructure of h which gives the same result. Formally, a non-redundant hybrid-solution $h = [\![s, \phi]\!]$ for $\mathcal{H}$ relative to Σ_h is a *minimal* hybrid-solution for $\mathcal{H}$ relative to Σ_h if there does not exist a non-redundant hybrid-solution $h' = [\![s', \phi']\!]$ for $\mathcal{H}$ relative to any subset of $sol(h', \mathcal{I}, \mathcal{D})$, where $s' \subset s$ and ϕ' is obtained from ϕ by replacing with $true$ every (ordering) constraint that mentions some task label occurring in the set $s \setminus s'$.

Thus, minimality is a stronger notion than non-redundancy. This was illustrated by the table in the previous example with hybrid-solutions $t_1 \cdot t_2$ and $t_1 \cdot t_3 \cdot t_4$, which were non-redundant but not minimal. We do not define minimality and non-redundancy as independent notions because that will then allow a hybrid-solution that is non-redundant and non-minimal, but where all minimal hybrid-solutions that can be obtained from it are redundant, as shown below.

[7] A subsequence is obtained from a given sequence by deleting zero or more elements from it and not changing the order of the remaining elements.

Example 6. Let us assume that minimality is defined relative to hybrid-solutions as opposed to non-redundant hybrid-solutions: i.e., a minimal hybrid-solution is a hybrid-solution from which no tasks can be removed to obtain a hybrid-plan that is still a hybrid-solution. Then, consider hybrid-solution $t_0 \cdot t_1$ for the hybrid planning problem having initial state $\{p, u\}$, goal condition s, and the HTN domain shown below. Observe that this hybrid-solution is non-redundant and non-minimal: it is non-redundant because it can produce the non-redundant primitive solution $a_1 \cdot a_2 \cdot a_3$ by selecting methods m_0 and m_2, and it is non-minimal because its proper subsequence t_0 is also a hybrid-solution—t_0 can produce the primitive solution $a_1 \cdot a_2 \cdot a_3 \cdot a_4$, by selecting method m_1. However, while hybrid-solution t_0 is minimal, it is also redundant, as its (only) primitive solution $a_1 \cdot a_2 \cdot a_3 \cdot a_4$ contains the redundant primitive task a_4.



Primitive Task	Prec.	Eff.
a_1	p	q
a_2	q	r
a_3	r	s
a_4	u	v

■

4.2 Maximally-Abstract Hybrid-Plans

As discussed earlier, a hybrid-plan is maximally-abstract if it does not match a refinement of any other hybrid-plan, where a refinement is an “intermediate” task network encountered via one or more decompositions of the given plan. To define a refinement, we extend the HTN semantics of [19], specifically their construct *sol*$(d, \mathcal{I}, \mathcal{D})$ (see Section 2.2) which represents the set of primitive plan solutions for d, so that intermediate task networks encountered via decompositions of d are also included in the set, in addition to the primitive ones.

Informally, the *refinements* of a task network d are the set of all task networks obtained by reducing d zero or more times; a single *refinement* is a member of this set. Formally, the set of refinements of a task network d is the reflexive transitive closure of the set of HTN reductions *red*$(d, \mathcal{D})$. In our definition below, *refn*$(d, \mathcal{D})^n$ is the set of task networks obtained from n reductions of d, and *refn*$(d, \mathcal{D})$ is the set of task networks obtained from all finite reductions of d.

Definition 5 (Refinements). The set of *refinements* of a task network d relative to an HTN domain $\mathcal{D}$, denoted by *refn*$(d, \mathcal{D})$, is defined as

$$\begin{aligned}
\mathit{refn}(d, \mathcal{D}) &= \bigcup_{n \in \mathbb{N}_0} \mathit{refn}^n(d, \mathcal{D}); \\
\mathit{refn}^0(d, \mathcal{D}) &= \{d\}; \\
\mathit{refn}^{n+1}(d, \mathcal{D}) &= \bigcup_{d' \in \mathit{refn}^n(d, \mathcal{D})} \mathit{red}(d', \mathcal{D}). \quad \blacksquare
\end{aligned}$$

Since a refinement of a task network is *any* “intermediate” task network d encountered during the decomposition of the former, there is no guarantee d will actually yield a primitive plan solution, i.e., it may be that *sol*$(d, \mathcal{I}, \mathcal{D}) = \emptyset$ for any state $\mathcal{I}$.

Given a refinement, we determine whether it *matches* a given hybrid-plan by basically checking whether the following conditions hold: *(i)* they have the same tasks in common; *(ii)* the ordering constraints in the hybrid-plan are compatible with those in the refinement;

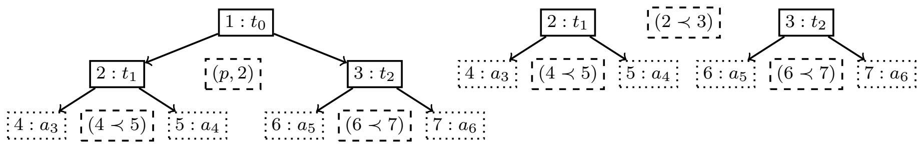


Fig. 5: Refinements depicted for hybrid-solution $[\![\{(1 : t_0)\}, \mathit{true}]\!]$ (left) and hybrid-solution $[\![\{(2 : t_1), (3 : t_2)\}, (2 \prec 3)]\!]$ (right). Dashed rectangles represent constraints on adjacent labelled tasks.

and *(iii)* the refinement and hybrid-plan have at least one primitive plan solution in common. This final check is necessary because the constraint formula of a hybrid-plan can only mention ordering constraints, whereas the constraint formula of a task network can be more "complex": it can additionally mention state and variable binding constraints, which may well make it more constrained than the hybrid-plan, and thereby incapable of producing any of the primitive solutions that the former can produce. This issue is illustrated below.

Example 7. Figure 5 illustrates the reductions of two hybrid-plans: $h = [\![\{(2 : t_1), (3 : t_2)\}, (2 \prec 3)]\!]$ on the right, and $h' = [\![\{(1 : t_0)\}, \mathit{true}]\!]$ on the left. Observe that the task network $d = [\![\{(2 : t_1), (3 : t_2)\}, (p, 2)]\!]$ shown in the figure is a refinement of h'—the former is obtained after performing a single reduction on the latter—and that the ordering constraint $(2 \prec 3)$ of hybrid-plan h is compatible with those in d.

Suppose $\neg p$ holds in the initial state, and that we wish to determine whether the refinement d of h' matches h relative to the set $\{(4 : a_3) \cdot (5 : a_4) \cdot (6 : a_5) \cdot (7 : a_6)\}$ containing the only primitive plan solution for h. Although the ordering constraints of h are compatible with those in refinement d, and h and d do mention the same tasks, the refinement nonetheless cannot produce the primitive plan solution above, as the state constraint $(p, 2)$ of d is violated with respect to the initial state. Thus, h' does not have a refinement that matches h, and h is indeed maximally-abstract. However, if p holds in the initial state, then h is not maximally-abstract, as refinement d of hybrid-plan h' does match h. ■

Using the above notions, we formally define the notion of a maximally-abstract hybrid-plan as one which does not match a refinement of any other hybrid-plan.

Definition 6 (Maximally-Abstract). Let $\mathcal{D}$ be an HTN domain, $\mathcal{I}$ a state, Δ a set of hybrid-plans, and $h = [\![s_h, \phi_h]\!] \in \Delta$ a hybrid-plan in the set. Finally, let $\Sigma_h \subseteq sol(h, \mathcal{I}, \mathcal{D})$ be a finite set of primitive plan solutions. Then, hybrid-plan h is *maximally-abstract* among set Δ for Σ_h if there exists no hybrid-plan $h' = [\![s_{h'}, \phi_{h'}]\!] \in \Delta$ with $|s_{h'}| < |s_h|$ such that:

1. $d_1 \in \mathit{refn}(h', \mathcal{D})$,
2. $d_2 = [\![s_{d_2}, \phi_{d_2}]\!]$ is a task label renaming of some $d_1\theta$ such that $s_{d_2} \supseteq s_h$,
3. $d_3 = [\![s_{d_2}, \phi_{d_2} \wedge \phi_h]\!]$, and
4. $\Sigma_h \cap sol(d_3, \mathcal{I}, \mathcal{D}) \neq \emptyset$.[8] ■

In words, a hybrid-plan h is maximally-abstract among hybrid-plans in Δ for a set of primitive plan solutions Σ_h if there is no shorter hybrid-plan h' in Δ that can produce h via decompositions, without losing *all* of the primitive plan solutions in Σ_h.

[8] If $\Sigma_h \subseteq sol(d_3, \mathcal{I}, \mathcal{D})$ also holds, h could be deemed *weakly maximally-abstract* among Δ for Σ_h; however, we only use the stronger notion in this paper.

4.3 Ideal Hybrid-Plans

An ideal hybrid-plan, then, is one that is minimal—and thereby non-redundant—as well as maximally-abstract relative to its set of perfect justifications.

Definition 7 (Ideal Hybrid-Plans). A given hybrid-plan h is an *ideal* one for a hybrid planning problem $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ if the following conditions hold:

1. h is a minimal hybrid-solution for $\mathcal{H}$ relative to $\Sigma \cap sol(h, \mathcal{I}, \mathcal{D})$, where Σ is the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$; and
2. h is a maximally-abstract hybrid-plan among *all* hybrid-plans for $\Sigma \cap sol(h, \mathcal{I}, \mathcal{D})$.

The set of all ideal hybrid-plans for $\mathcal{H}$ is denoted $ideal(\mathcal{H})$. ■

The following theorem states that whenever a hybrid planning problem can be solved, there is at least one ideal hybrid-plan.

Theorem 1. *Let $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ be a hybrid planning problem. If $sol(\mathcal{I}, \mathcal{G}, Op) \neq \emptyset$, then there exists an ideal hybrid-plan for $\mathcal{H}$.*

Proof. Let Σ^{nr} be the set of all perfect justifications for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$, let $\sigma \in sol(\mathcal{I}, \mathcal{G}, Op)$ be a primitive solution, and let σ' be a subsequence of σ such that $\sigma' \in \Sigma^{nr}$. Finally, let $h^1 = [\![\{(i : act_i) \mid act_i \in \sigma'\}, true \wedge \bigwedge\{(i \prec j) \mid i, j \in \{1, \ldots, |\sigma'|\}, i < j\}]\!]$ be the hybrid-plan representing σ' (recall that hybrid-plans cannot mention state constraints). There are two cases to consider. The first is that $\mathcal{I} \models \mathcal{G}$. In this case the theorem holds trivially because $|\sigma'| = 0$ and $h^1 = [\![\emptyset, true]\!]$ is an ideal hybrid-plan for $\mathcal{H}$. The second case is where $\mathcal{I} \not\models \mathcal{G}$. Then, hybrid-plan h^1 is a minimal hybrid-solution for $\mathcal{H}$ relative to $\{\sigma'\}$: it is minimal because there is no "subset" of h^1 that is still a hybrid-solution for $\mathcal{H}$, and it is non-redundant because σ' is non-redundant for $\langle \mathcal{I}, \mathcal{G}, Op \rangle$. Now, if h^1 is not a maximally-abstract hybrid-plan among all possible hybrid-plans for the set $\{\sigma'\}$, then by Definition 6 (Maximally-Abstract), a more abstract hybrid-plan exists, i.e., $MA(h^2, h^1, \{\sigma'\})$ holds for some hybrid-plan h^2, where $MA(h', h, \Sigma)$—for any hybrid-plans $h = [\![s_h, \phi_h]\!]$ and $h' = [\![s_{h'}, \phi_{h'}]\!]$, and set of primitive plans $\Sigma \subseteq sol(h, \mathcal{I}, \mathcal{D})$—denotes the conditions given in Definition 6 (that is, $|s_{h'}| < |s_h|, d_1 \in refn(h', \mathcal{D})$, etc.).

Let $min(h)$ denote the set of minimal hybrid-solutions $h_{min} = [\![s_{min}, \phi_{min}]\!]$ for $\mathcal{H}$ relative to $\Sigma^{nr} \cap sol(h_{min}, \mathcal{I}, \mathcal{D})$ that can be obtained from a hybrid-plan $h = [\![s_h, \phi_h]\!]$ (i.e., for any $[\![s_{min}, \phi_{min}]\!] \in min(h)$, the set $s_{min} \subseteq s_h$, and constraint ϕ_{min} is obtained from ϕ_h by replacing with $true$ every constraint that mentions some task label occurring in the set $s_h \setminus s_{min}$). Then, if there exists a hybrid-plan $h^3 \in min(h^2)$ such that h^3 is a maximally-abstract hybrid-plan among all possible hybrid-plans for the set $\Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$, we know that h^3 is an ideal hybrid-plan for $\mathcal{H}$, and the theorem holds. Otherwise, for each hybrid-plan $h^3 = [\![s^3, \phi^3]\!] \in min(h^2)$, there must exist a hybrid-plan h^4 such that $MA(h^4, h^3, \Sigma^3)$ holds, where $\Sigma^3 = \Sigma^{nr} \cap sol(h^3, \mathcal{I}, \mathcal{D})$. Observe that this reasoning can be continued for $h^4 = [\![s^4, \phi^4]\!]$ as we did before for h^2. However, since $|s^4| < |s^3|$ holds according to our definition of $MA(h^4, h^3, \Sigma^3)$, this reasoning can only be applied a *finite* number of times, until some hybrid-plan h^n, with $h^n = [\![\{(n : t)\}, true]\!]$ is reached for some compound task t. Hybrid-plan h^n, then, is a minimal hybrid-solution for $\mathcal{H}$ relative to the subset $\Sigma^{nr} \cap sol(h^n, \mathcal{I}, \mathcal{D})$, and also a maximally-abstract hybrid-plan among all possible hybrid-plans for the subset. □

Unfortunately, it is not clear how one could practically compute an ideal hybrid-plan for a hybrid planning problem.[9] In the next sections, we shall develop, and show how to implement, a weaker notion than an ideal plan, which looks for the most "preferred" specialisation of a fixed hybrid-plan.

5 Preferred Specialisations of Hybrid-Plans

Instead of searching for an ideal hybrid-solution, we now focus on "improving" one that is supplied, by exploring only the limited set of specialisations inherent in *one* of the hybrid-solution's decomposition traces in order to extract a most abstract and non-redundant specialisation. As before, the particular hybrid-solution that we start from may have been produced by a classical planner operating in the HTN domain, or by another technique.

The problem we wish to solve is as follows: *given a possibly redundant hybrid-solution h for a hybrid planning problem, along with a decomposition trace yielding one of h's primitive plan solutions that solves the problem, find a specialisation of h that is non-redundant and maximally abstract, but within the trace provided*. We call such specialisations *preferred* ones. While a preferred specialisation is not necessarily an ideal hybrid-plan as defined in Section 4, whenever an ideal hybrid-plan does occur within a decomposition trace of a given hybrid-plan h, the former plan will be a preferred specialisation of h.

Intuitively, a *decomposition trace* is a trace of the reductions performed on a hybrid-plan. Therefore, any hybrid-plan h that yields a primitive plan solution $\sigma \in sol(h, \mathcal{I}, \mathcal{D})$ will have at least one associated decomposition trace, starting from h and ending with a primitive task network that yields σ, as illustrated below. We call such decomposition traces *successful* decomposition traces.

Example 8. Consider a method-library allowing the following reductions: *(i)* task t_1 into labelled tasks $(2 : t_2)$ and $(3 : t_3)$ together with constraint $true$; *(ii)* task t_2 into labelled primitive tasks $(4 : a_4)$ and $(5 : a_5)$ together with constraint $4 \prec 5$; and *(iii)* task t_3 into $(6 : a_6)$ and $(7 : a_7)$ together with constraint $6 \prec 7$. Then, one possible decomposition trace of task network (hybrid-plan) $[\![\{(1 : t_1)\}, true]\!]$ is the following:[10]

$[\![\{(1 : t_1)\}, true]\!] \to [\![\{(2 : t_2), (3 : t_3)\}, true]\!] \to [\![\{(4 : a_4), (5 : a_5), (3 : t_3)\}, (4 \prec 5)]\!] \to [\![\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]\!]$. ■

While decomposition traces are intrinsic to the notion of a hybrid-solution, and thus necessary for the main results in this section, it is more intuitive to represent a trace as a *decomposition tree*, depicting how compound tasks were decomposed into more specific tasks and constraints. For example, Figure 6 shows the decomposition tree induced by the trace above. Observe that each node in the tree is a labelled task; each node is labelled with the HTN constraints enforced on its child nodes; and the children of the root node are the labelled tasks in the given hybrid-plan. While decomposition traces encode a specific order of task decompositions, decomposition trees are agnostic on *when* tasks are reduced. Thus, different traces may induce the same tree up to renaming of task labels. This is illustrated by the trace in Example 8 and Example 9 below, both of which induce the tree in Figure 6.

[9] From Theorem 1 it follows that checking whether there exists an ideal hybrid-plan for a hybrid planning problem is in **PSPACE**, as it amounts to checking for the existence of a (primitive) solution for the corresponding classical planning problem. However, as we currently do not have a better technique to *find* an ideal hybrid-plan than one involving a (naive) brute-force enumeration and check (which may require exponential space), we focus on a more restricted notion than an ideal hybrid-plan for which we can envision a technique.

[10] For readability we sometimes use notation $A \to B$ instead of $A \cdot B$ when referring to a sequence.

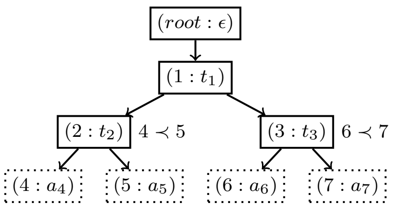


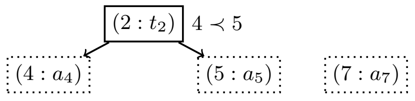
Fig. 6: The decomposition tree corresponding to decomposition trace $[\![\{(1 : t_1)\}, true]\!] \rightarrow [\![\{(2 : t_2), (3 : t_3)\}, true]\!] \rightarrow [\![\{(4 : a_4), (5 : a_5), (3 : t_3)\}, (4 \prec 5)]\!] \rightarrow [\![\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]\!]$. Missing constraint formulas stand for *true*.

Example 9. In the trace in Example 8, task t_1 is reduced first, t_2 second, and t_3 third. If instead t_3 is reduced before t_2, we would then have the following trace:

$[\![\{(1 : t_1)\}, true]\!] \rightarrow [\![\{(2 : t_2), (3 : t_3)\}, true]\!] \rightarrow [\![\{(2 : t_2), (6 : a_6), (7 : a_7)\}, (6 \prec 7)]\!] \rightarrow [\![\{(4 : a_4), (5 : a_5), (6 : a_6), (7 : a_7)\}, (4 \prec 5) \wedge (6 \prec 7)]\!]$. ■

Using the notions of a decomposition trace and tree, we can now reformulate the problem that we intend to solve in more precise terms as follows. Given a hybrid-plan h for a hybrid planning problem $\mathcal{H}$, and a decomposition tree $\mathcal{T}$ induced by a successful decomposition trace of h, we want to find a *forest* $\mathcal{T}'$ within $\mathcal{T}$ such that $\mathcal{T}'$ yields a perfect justification but $\mathcal{T}'$ is not subsumed by some other forest within $\mathcal{T}$ that also yields the leaf-level nodes of $\mathcal{T}'$. We can then take the hybrid-plan corresponding to the root nodes of $\mathcal{T}'$ as a preferred specialisation of h for $\mathcal{T}$ and $\mathcal{H}$. This process is illustrated below.

Example 10. Suppose the decomposition trace shown in the above example is a successful trace of hybrid-plan $h = [\![\{(1 : t_1)\}, true]\!]$, and that plan $(4 : a_4) \cdot (5 : a_5) \cdot (7 : a_7)$ (a subsequence of one of the trace's associated primitive plans) is a perfect justification for some problem $\mathcal{H}$—i.e., labelled task $(6 : a_6)$ is redundant. Then, a preferred specialisation of h for $\mathcal{H}$, relative to the tree in Figure 6, is the hybrid-plan $[\![\{(2 : t_2), (7 : a_7)\}, true]\!]$ represented by the root nodes of the forest depicted below. Observe that *(i)* the forest occurs within the tree in Figure 6; *(ii)* the forest yields the aforementioned perfect justification; and that *(iii)* there is no other forest occurring within the tree in Figure 6 that is more abstract than the forest below.[11]



■

In order to develop a formal account of a preferred specialisation, we shall formalise the notions of decomposition traces and trees, define what it means for a tree to be *executable* in a state, and finally, what it means for one such tree to *dominate* another. Informally, a decomposition tree is executable in a state if all constraints (labels) associated with the tree's nodes are satisfied, and the tree's leaf-level nodes (primitive tasks) are executable. Given two forests within an executable decomposition tree, one forest dominates the other if the former "contains" the latter and yields the same leaf-level nodes.

[11] Note that the forest with root nodes $(2 : t_2)$ and $(3 : t_3)$ is not more abstract than the forest in this example because the former forest does not yield the same leaf-level nodes.

5.1 Decomposition Traces and Trees

Formally, a *decomposition trace* of a task network (or hybrid-plan) is a sequence of ground task networks, where each task network d_i in the sequence is a reduction of a task in the preceding task network d_{i-1}. Such a sequence may be infinite if recursive compound tasks occur in methods (a recursive compound task is shown in Example 2).

Definition 8 (Decomposition Trace). Let $\mathcal{D} = \langle Op, Me \rangle$ be an HTN domain, $\mathcal{I}$ an initial state, and d a task network. A *decomposition trace* of d relative to *Me* is a possibly infinite sequence of ground task networks $\lambda = d_1 \cdot \ldots \cdot d_n \cdot \ldots$, such that $d_1 = d\theta$ (for some substitution θ), and for each $d_i, i > 0$, it holds that: *(i)* $d_{i+1} = reduce(d_i, n, m)$, where n is a task label occurring in d_i and m is a ground instance of a method in *Me*; and *(ii)* there is no common task label occurring in $s_{i+1} \setminus s_i$ and $d_1 \cdot \ldots \cdot d_i$ (where each d_j is of the form $[\![s_j, \phi_j]\!]$). A finite decomposition trace $d_1 \cdot \ldots \cdot d_n$ of d relative to *Me* is *complete* if d_n is a primitive task network. Finally, a complete decomposition trace $d_1 \cdot \ldots \cdot d_n$ of d relative to *Me* is *successful* relative to $\mathcal{I}$ and $\mathcal{D}$ if $sol(d_n, \mathcal{I}, \mathcal{D}) \neq \emptyset$. ■

We highlight three important subtleties in this definition. First, a decomposition trace encodes a specific order on the reduction of tasks. Second, all tasks added into a trace via reduction—i.e., the set $s_{i+1} \setminus s_i$—have different labels to those that already occur in the trace, ensuring that task labels are unique across the trace. Third, the last task network in a complete decomposition trace may have tasks for which no ordering is specified, and thereby yield more than one primitive plan solution. The trace in Example 9 ends with such a task network, where tasks a_5 and a_6 are not ordered.

Next, we define an *induced decomposition tree*. Recall that this simply represents a decomposition trace in the form of a tree, depicting how compound tasks are reduced via methods into child tasks and their associated constraints. Specifically, a node of an induced decomposition tree is a labelled (compound or primitive) task, and each node is labelled with the constraints (if any) that are enforced on its child tasks; root nodes and those representing ϵ (dummy) tasks are distinguished by the ϵ symbol.[12] The definitions that follow rely on notions that center upon a *vertex-labelled tree*, which are listed in Appendix C.

Definition 9 (Induced Decomposition Tree). Let $\lambda = [\![s_1, \phi_1]\!] \cdot \ldots \cdot [\![s_k, \phi_k]\!]$ be a decomposition trace of some task network relative to a method-library *Me*. Suppose V_λ is the set of labelled tasks mentioned in λ, and $rt = (root : \epsilon)$. The *induced decomposition tree* of λ, then, is the vertex-labelled tree $\langle V_\lambda \cup \{rt\}, E, \ell_V \rangle$, where

$$
\begin{aligned}
E &= \bigcup_{u \in s_1} \Big(\{(rt, u)\} \cup edges^*(u) \Big), \\
&\quad edges^*(u) = edges(u) \cup \bigcup_{(u,u') \in edges(u)} edges^*(u'), \\
&\quad edges(u) \;\; = \{(u, u') \mid 0 < i < k,\ u' \in (s_{i+1} \setminus s_i),\ u \in s_i,\ u \notin s_{i+1}\}; \text{ and} \\
\ell_V &= \{(rt, \phi_1)\} \cup \\
&\quad \bigcup_{u \in V_\lambda} \{(u, \phi) \mid 0 < i < k,\ u \in s_i,\ u \notin s_{i+1},\ \phi_{i+1} = \phi_i \wedge \phi\} \cup \\
&\quad \bigcup_{u \in V_\lambda} \{(u, true) \mid (u, u') \notin E\}.^{13} \;\blacksquare
\end{aligned}
$$

12 Recall from Section 2.3 that it is sometimes useful to have a method consisting of a single ϵ task. Such a method would amount to "doing nothing."

A (general) *decomposition tree* $\mathcal{T}$ of a task network d relative to a method-library *Me* is the induced decomposition tree of *some* decomposition trace of d relative to *Me*. As in the definition of a decomposition trace (Definition 8), the set $s_{i+1} \setminus s_i$ above is the set of labelled tasks added by the reduction of labelled task u. Observe that the label of the root node is the constraint formula ϕ_1 in the first task network of λ, and that leaf nodes have label *true*. Definition 9 is a generalisation of the definition of a decomposition tree in [24], which concerns trees that are not associated with state and variable binding constraints.

5.2 Executable Decomposition Trees

So far, decomposition trees have been merely *syntactic* objects independent from states—the former describe legal syntactic ways of transforming tasks into other tasks with respect to the method-library. We shall now combine decomposition trees with states to define what it means for a decomposition tree to be *executable* in a state, with respect to a particular primitive plan yielded by the tree. To this end, we first define the following auxiliary notions: *complete decomposition tree*, *linearisation*, and *full decomposition tree*.

A decomposition tree (of a task network relative to a method-library) is *complete* if no leaf node mentions a compound task, i.e., a leaf node can only represent a primitive task or a node of the form $(n : \epsilon)$ (i.e., a dummy task). A *linearisation* τ of a complete decomposition tree $\mathcal{T}$ is a permutation of the elements in *leaves*$(\mathcal{T}) \setminus \{(root : \epsilon)\}$, i.e., a labelled primitive plan built from exactly the non-root elements in *leaves*$(\mathcal{T})$. For example, a linearisation of the tree in Figure 7 is the sequence $(2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (10 : a_6) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$. Notice that the ordering of elements in a linearisation is independent of any ordering constraints enforced on them in $\mathcal{T}$, and that a linearisation is independent of any state. Hence, while a linearisation of a hybrid-plan's complete decomposition tree is a (labelled) primitive plan, the linearisation is not necessarily a (labelled) primitive plan *solution* for the hybrid-plan (for some initial state). Finally, a *full decomposition tree*, denoted $\mathcal{T}_\tau$, is a tuple $\langle \mathcal{T}, \tau \rangle$, where $\mathcal{T}$ is a complete decomposition tree and τ is a linearisation of $\mathcal{T}$. Thus, a full decomposition tree encodes not just how tasks are completely reduced into primitive tasks, but also how these may be ordered to form a labelled primitive plan.

Then, we say that a full decomposition tree $\mathcal{T}_\tau$ is *executable* in an initial state $\mathcal{I}$ if the tree is "legal" in $\mathcal{I}$, i.e., all constraints occurring in the tree are satisfied in $\mathcal{I}$, and the labelled primitive plan τ is executable in $\mathcal{I}$. We demonstrate what it means for a constraint formula to be *satisfied* in a state (relative to the tree) with the following example; a formal definition can be found in Appendix A (Definition 13).

Example 11. Consider the full decomposition tree $\mathcal{T}_\tau$, where $\mathcal{T}$ is the complete decomposition tree in Figure 7 and linearisation $\tau = (2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (10 : a_6) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$. Suppose that we want to determine whether the constraint formula of node $(4 : t_2)$—i.e., $(5 \prec 8) \wedge (5 \prec 11)$—is satisfied relative to the full decomposition tree (the initial state is not needed in this example as we only deal here with ordering constraints). Observe that $(5 \prec 8)$ is indeed satisfied in τ because all primitives corresponding to task label 5—i.e., $(6 : a_3)$ and $(7 : a_4)$—precede

[13] Actually, ϕ_i will have to be modified so that all occurrences of the task label in u are replaced appropriately with expressions of the form *first*[] or *last*[], as done in the definition of a reduction (Definition 15). Note that since ℓ_V is a mapping from task labels to constraint formulas, we sometimes treat ℓ_V as a set of ordered pairs for convenience.

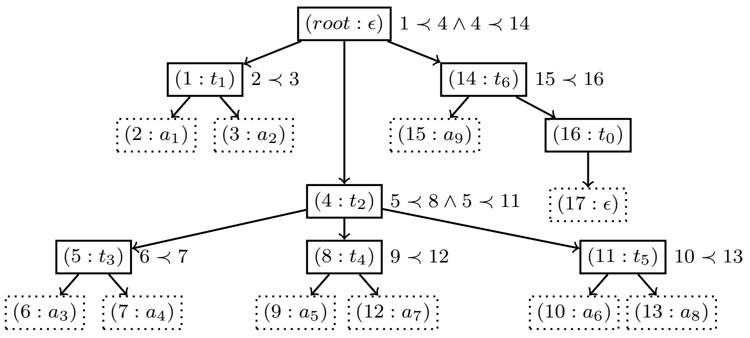


Fig. 7: A complete decomposition tree of task network $d = [\![\{(1 : t_1), (4 : t_2), (14 : t_6)\}, (1 \prec 4) \wedge (4 \prec 14)]\!]$. Node $(17 : \epsilon)$ represents an empty reduction.

all the primitives corresponding to task label 8—i.e., $(9 : a_5)$ and $(12 : a_7)$. However, constraint $(5 \prec 11)$ is not satisfied in τ because one of the primitives corresponding to task label 11—i.e., $(10 : a_6)$—does not precede all the primitives corresponding to task label 5. Consequently, formula $(5 \prec 8) \wedge (5 \prec 11)$ is also not satisfied. On the other hand, if instead we have $\tau = (2 : a_1) \cdot (3 : a_2) \cdot (6 : a_3) \cdot (7 : a_4) \cdot (9 : a_5) \cdot (12 : a_7) \cdot (10 : a_6) \cdot (13 : a_8) \cdot (15 : a_9) \cdot (17 : \epsilon)$, then the formula $(5 \prec 8) \wedge (5 \prec 11)$ is indeed satisfied. ■

Our formal definition of an executable (full) decomposition tree refers to the construct $Res^*(act_1 \cdot \ldots \cdot act_n, \mathcal{I}, Op)$ from Section 2.1, which denotes the state resulting from applying actions $act_1 \cdot \ldots \cdot act_n$ from the initial state $\mathcal{I}$.

Definition 10 (Executable Decomposition Tree). Let $\mathcal{T}_\tau$ be a full decomposition tree, $\mathcal{I}$ be an initial state, and let *Op* be an operator-library. Then, $\mathcal{T}_\tau$ is *executable* in $\mathcal{I}$ relative to *Op* if *(i)* for all $u \in V(\mathcal{T})$, constraint formula $\ell_V(u)$ is satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and *Op*, and *(ii)* $Res^*(\tau, \mathcal{I}, Op)$ is defined. ■

In what follows, we use $\langle \mathcal{T}_\tau, \mathcal{I}, Op \rangle \models \phi$ to denote that constraint formula ϕ is satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and *Op*, or we simply write $\mathcal{T}_\tau \models \phi$ when the operator-library is obvious from context, and the constraint formula does not mention any HTN state constraints (and consequently, the initial state is not needed to check whether ϕ is satisfied, as illustrated in the example above).[14]

The following lemma states that whenever there is a primitive plan solution associated with a complete decomposition trace, then that plan, combined with the trace's induced tree, amounts to an executable decomposition tree, and vice versa. The lemma relies on construct $comp(d, \mathcal{I}, \mathcal{D})$ (Definition 14), an element of which is a grounding and ordering of the primitive tasks in d that does not conflict with the constraints imposed on them in d. (Proofs for lemmas can be found in Appendix B.)

Lemma 1. *Let $\mathcal{D}$ be an HTN domain, $\lambda = d_1 \cdot \ldots \cdot d_k$ a complete decomposition trace of a task network relative to Me, and $\mathcal{T}$ the induced (complete) decomposition tree of λ. Then, there exists a plan $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$ if and only if the full decomposition tree $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to Op, with $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$.*

[14] We note that while the truth value of any constraint occurring in a *full* decomposition tree can be determined given the initial state, this is not necessarily the case for standard (non-full) decomposition trees, because the satisfaction of a constraint generally depends on the total-ordering chosen for the primitive tasks.

This lemma is a variant of the first proposition in [24], which concerns the existence of a trace that yields a task network whenever the latter can also be yielded by a tree. The above lemma is restricted to a tree whose yield consists of primitive tasks, and these have a linearisation that is executable in the initial state.

5.3 Preferred Specialisations

We have now provided most of the formal machinery needed for our final definition of a *preferred specialisation*. Recall that a preferred specialisation is one that is both non-redundant and as abstract as possible within the confines of a given decomposition trace of the hybrid-plan. In order to define a preferred specialisation we rely on some auxiliary notions centering upon *cuts* in a decomposition tree and *dominance* between cuts.

A *cut* in a decomposition tree, combined with the cut's *projection*, concretises our notion of a forest within a tree. Informally, a cut in a decomposition tree is a subset of its nodes which, together with their constraints, can form a hybrid-plan. It is important to ensure that nodes in a cut are not descendants of other nodes that are also in the cut; e.g., while the set of nodes $\{(1 : t_1), (4 : t_2)\}$ is a (legal) cut in the tree shown in Figure 7, the set $\{(4 : t_2), (5 : t_3)\}$ is not, because $(5 : t_3)$ is a descendant of $(4 : t_2)$.

Guided by the notion of maximal-abstraction (Definition 6), the notion of *dominance* intuitively states that some cuts are more abstract than others. For example, if we take cuts $\pi_1 = \{(4 : t_2)\}$ and $\pi_2 = \{(5 : t_3), (8 : t_4), (11 : t_5)\}$ in the tree shown in Figure 7, π_1 dominates π_2 because the latter occurs in the descendants of the former, and both π_1 and π_2 yield the same non-ϵ primitive tasks. More precisely, a cut π_1 dominates a cut π_2 if π_1, together with its descendants, contains π_2, and π_1 produces exactly the same non-ϵ primitive tasks as those produced by π_2. Observe that in general, whenever a cut π_1 dominates another cut π_2, any compound task that is a descendant of π_1 but not in π_2 nor its descendants can only yield ϵ tasks. Formally, the notion of a cut and of dominance is defined as follows.

Definition 11 (Cut & Dominance). A *cut* in a decomposition tree $\mathcal{T}$ is a set of nodes $\pi \subseteq V(\mathcal{T})$, with $\pi \neq \{(root : \epsilon)\}$, such that for all $u, u' \in \pi$, with $u \neq u'$, it is the case that $(descendants(u, \mathcal{T}) \cup \{u\}) \cap (descendants(u', \mathcal{T}) \cup \{u'\}) = \emptyset$. Given cuts π' and π in a decomposition tree $\mathcal{T}$, cut π' *dominates* π in $\mathcal{T}$ if $\pi \subseteq \bigcup_{u \in \pi'} descendants(u, \mathcal{T}) \cup \pi'$, and $actions(\mathcal{T}|_{\pi'}) = actions(\mathcal{T}|_{\pi})$.[15] ■

We use $\mathcal{T}|_\pi$ to denote the decomposition tree obtained by *projecting* on a cut $\pi \subseteq V(\mathcal{T})$ in the decomposition tree $\mathcal{T} = \langle V, E, \ell_V \rangle$, that is, the new tree $\mathcal{T}'$ obtained by projecting only on the nodes in π, trivially adding node $(root : \epsilon)$ as root with π as its children, and setting label $\ell_V((root : \epsilon))$ to $true$. For example, Figure 8 shows the projected tree for cut $\{(1 : t_1), (4 : t_2)\}$. The notion of projecting on a cut π trivially generalises to a full decomposition tree $\mathcal{T}_\tau$, which we denote as $\mathcal{T}_\tau|_\pi$. Formal definitions for projections are given in Appendix A (from Definition 19).

We can now define the notion of a preferred specialisation of a hybrid-plan. For convenience, we use the construct $decsol(h, \mathcal{H})$, where $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$, to denote the set of full decomposition trees $\mathcal{T}_\tau$ of a hybrid-plan h relative to a domain $\mathcal{D}$, where *(i)* $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to *Op*, and *(ii)* $\tau \in sol(\mathcal{I}, \mathcal{G}, Op)$, i.e., the linearisation τ achieves the

[15] Function $actions(\mathcal{T})$ denotes the set of non-ϵ nodes that are primitive in $\mathcal{T}$, i.e., $actions(\mathcal{T}) = \{(n : t) \mid (n : t) \in leaves(\mathcal{T}), t \neq \epsilon\}$.

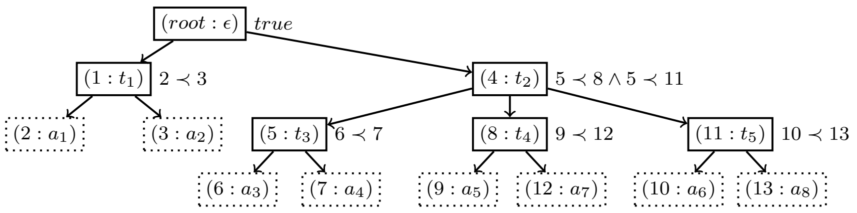


Fig. 8: The decomposition tree obtained from the tree in Figure 7 by projecting on cut $\{(1 : t_1), (4 : t_2)\}$.

goal condition $\mathcal{G}$. Moreover, given a cut π in a full decomposition tree $\mathcal{T}_\tau$, we define the ordering constraints implied by $\mathcal{T}_\tau$ on π as *true* if $\pi = \emptyset$, and otherwise as follows:[16]

$$\Phi[\mathcal{T}_\tau, \pi] = \bigwedge\nolimits_{\{n_1 \prec n_2 \mid (n_1:t_1),(n_2:t_2) \in \pi, \mathcal{T}_\tau \models n_1 \prec n_2\}} .$$

Then, given an executable (full) decomposition tree, a preferred specialisation is essentially a cut in it whose corresponding forest yields a perfect justification, and is not dominated by any other cut in the tree—i.e., the cut is as abstract as possible in the tree.

Definition 12 (Preferred Specialisation). Let $\mathcal{H}$ be a hybrid planning problem, h a hybrid-plan, and $\mathcal{T}_\tau \in \mathit{decsol}(h, \mathcal{H})$. Then, a hybrid-plan h_π is a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$ if $h_\pi = [\![\pi, \Phi[\mathcal{T}_\tau, \pi]]\!]$ for some cut π in $\mathcal{T}_\tau$ such that

1. the projected linearisation $\tau|_{\mathit{actions}(\mathcal{T}|_\pi)}$ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, \mathit{Op} \rangle$;
2. the projected full decomposition tree $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to Op; and
3. there is no cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op. ■

Note that a preferred specialisation h_π of a hybrid-plan h does not necessarily represent a *refinement* (i.e., an HTN decomposition) of h, as π may need to be a proper subset of a refinement's tasks in order to yield a perfect justification. Thus, h_π is a refinement-like "extraction", from tree $\mathcal{T}$, of an alternative for h that meets the above conditions. Recall that the motivation behind searching for a preferred specialisation is that finding an ideal hybrid-plan is likely to be more computationally expensive.

The third condition above ensures that a cut π' that contains π but also contains other compound tasks that lead to ϵ tasks is not preferred over π. Nonetheless, such a cut π' may well be a preferred specialisation—even if it does contain tasks that lead to ϵ tasks. Preferred specialisations that do not contain any compound tasks leading to ϵ tasks are called *minimal* preferred specialisations. Formally, a preferred specialisation $[\![s, \phi]\!]$ of a hybrid-plan h within a full decomposition tree $\mathcal{T}_\tau$ for a hybrid planning problem $\mathcal{H}$ is a minimal preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$ if there does not exist a preferred specialisation $[\![s' \subset s, \phi']\!]$ of h within $\mathcal{T}_\tau$ for $\mathcal{H}$, where ϕ' is obtained from ϕ by replacing all (ordering) constraints that mention some task label in $(s \setminus s')$ with $true$.

The following result guarantees that there is always a preferred specialisation for a hybrid-solution. More specifically, whenever an executable decomposition tree that achieves a given goal condition exists for a hybrid-plan, a preferred specialisation also exists for it.

Theorem 2. *Let $\mathcal{H}$ be a hybrid planning problem, and let h be a hybrid-plan. If $\mathcal{T}_\tau \in \mathit{decsol}(h, \mathcal{H})$, then there exists at least one preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

[16] Note that $\mathcal{T}_\tau \models n_1 \prec n_2$ holds if all leaves of n_1 occur before all leaves of n_2 in τ.

Proof. Let $\mathcal{T}_\tau \in \mathit{decsol}(h, \mathcal{H})$ be any full decomposition tree (recall $\tau \in \mathit{sol}(\mathcal{I}, \mathcal{G}, \mathit{Op})$ is a labelled primitive solution for $\mathcal{H}$). Then, it follows from the definition of a decomposition tree and Lemma 1 that $\tau \in \mathit{sol}(h, \mathcal{I}, \mathcal{D})$ is a primitive plan solution for h. Let τ' be a subsequence of τ that is a perfect justification for $\mathcal{C} = \langle \mathcal{I}, \mathcal{G}, \mathit{Op}\rangle$, and π be the "low-level" cut of τ', i.e., $\pi = \{(n : t) \mid (n : t) \in \tau'\}$. There are two cases to consider; the first is that $\mathcal{I} \models \mathcal{G}$. Then, $\pi = \emptyset$ and hybrid-plan $h_\pi = [\![\emptyset, \mathit{true}]\!]$ is indeed a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

The second case to consider is where $\mathcal{I} \not\models \mathcal{G}$. Then, let hybrid-plan $h_\pi = [\![\pi, \Phi[\mathcal{T}_\tau, \pi]]\!]$. Observe that the projections $\tau|_{\mathit{actions}(\mathcal{T}|_\pi)} = \tau'$ yield a perfect justification for $\mathcal{C}$, and thereby that the first condition in Definition 12 (Preferred Specialisation) holds for hybrid-plan h_π to be deemed a preferred specialisation (of hybrid-plan h within $\mathcal{T}_\tau$ for $\mathcal{H}$). The second condition in the definition also holds for h_π because the label (constraint formula) of the root node in $\mathcal{T}_\tau|_\pi$ is *true*, and $\mathit{Res}^*(\tau', \mathcal{I}, \mathit{Op})$—the state resulting from applying τ' in $\mathcal{I}$—is defined, and therefore $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to Op (Definition 10).

Now, suppose that h_π does not meet the third condition in Definition 12, i.e., there does exist a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op. Then, since π' dominates π in $\mathcal{T}$, it must hold, according to Definition 11 (Dominance), that $\mathit{actions}(\mathcal{T}|_{\pi'}) = \mathit{actions}(\mathcal{T}|_\pi)$, and therefore, that $\tau|_{\mathit{actions}(\mathcal{T}|_{\pi'})} = \tau'$ is also a perfect justification for $\mathcal{C}$. Given that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to Op, if hybrid-plan $[\![\pi', \Phi[\mathcal{T}_\tau, \pi']]\!]$ is still not a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$, then there must exist yet another cut π'' in $\mathcal{T}$, with $|\pi''| < |\pi'|$, that dominates π' in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi''}$ is executable in $\mathcal{I}$ relative to Op. Observe that this reasoning can be continued for π'' like we did before for cut π'. However, since $|\pi''| < |\pi'|$, this reasoning can only be applied a *finite* number of times, until some cut $\pi^{top} \subseteq \mathit{children}((\mathit{root} : \epsilon), \mathcal{T})$ is reached. Such a cut will have no strict subset that can dominate it in $\mathcal{T}$, and hybrid-plan $[\![\pi^{top}, \Phi[\mathcal{T}_\tau, \pi^{top}]]\!]$ will therefore be a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$. □

Since the dominance relation among cuts is not total, and there may well be more than one perfect justification that can be obtained from a linearisation, there may also be more than one preferred specialisation for a hybrid-plan within its executable decomposition tree.

Recall that our ideal hybrid-plan (Definition 7) defined a non-redundant plan that is maximally-abstract *among all conceivable hybrid-plans*. On the other hand, a preferred specialisation is non-redundant and maximally abstract among only the hybrid-plans that occur within a particular decomposition tree of a given hybrid-plan. Thus, whenever an ideal hybrid-plan does occur within such a tree, the former must also be a preferred specialisation of the given hybrid-plan. This relationship is concretised by the next theorem. In what follows, we use $\mathit{decsol}^{nr}(h, \mathcal{H}) \subseteq \mathit{decsol}(h, \mathcal{H})$ to denote the set of executable trees $\mathcal{T}_\tau$ where $\tau|_{\mathit{actions}(\mathcal{T})}$ is a perfect justification for problem $\langle \mathcal{I}, \mathcal{G}, \mathit{Op}\rangle$.[17]

Theorem 3. *Let h be a hybrid-plan, $\mathcal{H}$ a hybrid planning problem, and $\mathcal{T}_\tau \in \mathit{decsol}(h, \mathcal{H})$. Suppose there exists a cut π in $\mathcal{T}_\tau$ such that $h_\pi = [\![\pi, \Phi[\mathcal{T}_\tau, \pi]]\!] \in \mathit{ideal}(\mathcal{H})$ and such that there exists a full tree $\mathcal{T}^\pi_{\tau^\pi} \in \mathit{decsol}^{nr}(h_\pi, \mathcal{H})$ that is equivalent to $\mathcal{T}_\tau|_\pi$, modulo their root nodes. Then, hybrid-plan h_π is a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

Proof. Let $\sigma = \tau|_{\mathit{actions}(\mathcal{T}^\pi)}$ be the projected linearisation of actions representing cut π. Then, observe that the following three conditions hold:

(a) σ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, \mathit{Op}\rangle$ (because $\mathcal{T}^\pi_{\tau^\pi} \in \mathit{decsol}^{nr}(h_\pi, \mathcal{H})$);

[17] Recall that $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D}\rangle$ and $\mathcal{D} = \langle \mathit{Op}, \mathit{Me}\rangle$.

(b) $\sigma \in sol(h_\pi, \mathcal{I}, \mathcal{D})$ follows from our first assumption in Section 2.3, the definition of a decomposition tree, and Lemma 1; and
(c) the projected full tree $\mathcal{T}_\tau|_\pi$ is executable in $\mathcal{I}$ relative to *Op* (because this holds for $\mathcal{T}^\pi_{\tau^\pi}$, which is equivalent to $\mathcal{T}_\tau|_\pi$ modulo their root nodes).

Due to (a) and (c) above, the first two conditions in the definition of a preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$ (Definition 12) are met by hybrid-plan h_π. Next, we prove by contradiction that the third condition in the definition is also met by hybrid-plan h_π.

Let us assume the contrary, i.e., that there does exist a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to *Op*. We shall show that assuming this implies $h_\pi \notin ideal(\mathcal{H})$, which contradicts an assumption of the theorem. Observe from Definition 6 (Maximally-Abstract) and condition (b) above that to show $h_\pi \notin ideal(\mathcal{H})$, it is sufficient to show that the following four conditions hold for $h_{\pi'} = [\![\pi', true]\!]$ (we take $h' = h_{\pi'}$ and $h = h_\pi$ for Definition 6):

$$d_1 \in refn(h_{\pi'}, \mathcal{D}); \tag{1}$$
$$d_2 = [\![s_{d_2}, \phi_{d_2}]\!] \text{ is a ground instance of } d_1 \text{ such that } s_{d_2} \supseteq \pi; \tag{2}$$
$$d_3 = [\![s_{d_2}, \phi_{d_2} \wedge \Phi[\mathcal{T}_\tau, \pi]]\!]; \text{ and} \tag{3}$$
$$\sigma \in sol(d_3, \mathcal{I}, \mathcal{D}). \tag{4}$$

Equations (1) and (2) rely on the following two facts. First, given that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to *Op*, it follows that there is a (complete and ground) decomposition trace $d'_1 \cdot \ldots \cdot d'_k$ of $h_{\pi'}$ relative to *Me*, with $d'_1 = h_{\pi'}$, such that $\mathcal{T}' = \mathcal{T}|_{\pi'}$ is the induced (complete and ground) decomposition tree of the trace. Second, as illustrated in Example 9, changing the order in which reductions are performed on task networks in a decomposition trace will still yield the same primitive task network.[18] Then, since π' dominates π and both of these are (valid) cuts in $\mathcal{T}$, it is not difficult to see that there is a decomposition trace of $h_{\pi'}$ (relative to *Me*) that "goes through" π, namely the trace

$$d'_1 = [\![s_1, \phi_1]\!] \cdot \ldots \cdot [\![s_j, \phi_j]\!] \cdot \ldots \cdot d'_k = [\![s_k, \phi_k]\!],$$

where *(i)* $s_1 = \pi'$ and $\phi_1 = true$; *(ii)* $\pi \subseteq s_j$; and *(iii)* $j \in \{2, \ldots, k\}$ ($j \neq 1$ because $|\pi| > |s_1|$). Finally, since it follows from Definitions 5 and 8 that task network $[\![s_j, \phi_j]\!]$ is simply a ground instance of some refinement in $refn(h_{\pi'}, \mathcal{D})$, and since $\pi \subseteq s_j$ holds, Equations (1) and (2) also hold.

We shall now show that Equations (3) and (4) also hold. Let $\sigma' = \tau|_{actions(\mathcal{T}')}$. Since π' dominates π in $\mathcal{T}$, we know that $\sigma = \sigma'$ (recall that $\sigma = \tau|_{actions(\mathcal{T}^\pi)}$). Moreover, since $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to *Op*, it follows as before that $\sigma \in sol([\![s_1, \phi_1]\!], \mathcal{I}, \mathcal{D})$ holds, and from [16] that $\sigma \in sol([\![s_j, \phi_j]\!], \mathcal{I}, \mathcal{D})$ also holds.[19] Finally, due to condition (b) at the start of the proof, and since $\Phi[\mathcal{T}_\tau, \pi]$ does not "conflict" with ϕ_j, i.e., the former is simply the conjunction of ordering constraints entailed by σ on elements in π, it follows that $\sigma \in sol([\![s_j, \phi_j \wedge \Phi[\mathcal{T}_\tau, \pi]]\!])$ also holds. Therefore, Equations (3) and (4) hold, and h_π is not a maximally-abstract hybrid-plan (i.e., $h_\pi \notin ideal(\mathcal{H})$), which contradicts an assumption of the theorem. □

[18] Specifically, for a task network d that mentions two non-primitive tasks with labels n_1 and n_2, the task networks $reduce(reduce(d, n_1, m_1), n_2, m_2)$ and $reduce(reduce(d, n_2, m_2), n_1, m_1)$ for any methods m_1 and m_2 are equal up to variable and node label renaming [17].

[19] Specifically, inference rule **R2** in [16] states that if $d' \in red(d, \mathcal{I}, \mathcal{D})$ and $\sigma \in sol(d', \mathcal{I}, \mathcal{D})$, then conclude $\sigma \in sol(d, \mathcal{I}, \mathcal{D})$.

Algorithm 1 Find-Preferred-Specialisation$(h, \mathcal{H}, \mathcal{T}_\tau)$

Require: Hybrid-plan h, hybrid planning problem $\mathcal{H}$, $\mathcal{T}_\tau \in \mathit{decsol}(h, \mathcal{H})$, where $\mathcal{T} = \langle V, E, \ell_V \rangle$.
Ensure: A preferred specialisation of h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

```
1:  $\tau' \Leftarrow$ Get-Perfect-Justification$(\tau, \mathcal{H})$                 // As in [21]; ignore $\epsilon$ tasks
2:  $\pi \Leftarrow \{(n : t) \mid (n : t) \in \tau'\}$
3:  for $\ell \Leftarrow 1$ to $\mathit{height}(\mathcal{T}) - 1$ do                     // Leaves are at level $0$ (Definition 17)
4:      for each node $u$ at level $\ell$ in tree $\mathcal{T}$ do
5:          if $\mathit{children}(u, \mathcal{T}) \subseteq \pi$ and $\langle \mathcal{T}_\tau|_\pi, \mathcal{I}, \mathit{Op} \rangle \models \ell_V(u)$ then      // $\ell_V(u)$ is satisfied in $\mathcal{T}_\tau|_\pi$
6:              $\pi \Leftarrow (\pi \setminus \mathit{children}(u, \mathcal{T})) \cup \{u\}$          // Replace $u$'s children with $u$
7:          end if
8:      end for
9:  end for
10: $\pi \Leftarrow \pi \setminus \Delta \Leftarrow \{u \mid u \in \pi$, all leaves of $\mathcal{T}|_{\{u\}}$ are $\epsilon$ nodes$\}$
11: $\phi \Leftarrow \Phi[\mathcal{T}_\tau, \pi]$                                   // As defined just before Definition 12
12: return $[\pi, \phi]$
```

6 Computing Preferred Specialisations

Algorithm 1 shows how a preferred specialisation can be computed from an executable decomposition tree of a hybrid-plan h, given a hybrid planning problem $\mathcal{H}$. To obtain an executable decomposition tree, we do the following. First, we obtain a (standard) decomposition tree by finding any successful decomposition trace for h, and then inducing a tree from the trace as in Definition 9. We then create an executable decomposition tree by combining the induced tree with a (labelled) primitive plan solution associated with the trace. Finding a successful decomposition trace can be done with a trivial modification to the UMCP HTN algorithm [19] so that it keeps track of all reductions performed on tasks, leaves the labels of primitive tasks intact, and checks that primitive plans achieve a given goal condition.[20]

Basically, Algorithm 1 works bottom-up, by starting at the leaf-level with a labelled primitive plan that is a perfect justification (line 1), and then repetitively abstracting one or more steps into a higher-level more abstract step (lines 3-9). Once abstracting is no longer possible, the ordering constraints entailed by the decomposition tree on the final set of tasks are calculated (line 11) and the final hybrid-plan is returned. Next, we describe certain lines in the algorithm in more detail.

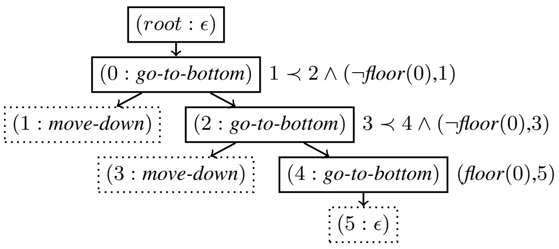
In line 1, we ignore ϵ tasks when looking for a perfect justification because otherwise they will be considered redundant, and will thereby be removed from τ. While ϵ tasks are indeed redundant by definition (as they have no effects), they are still distinct from "real" redundant primitive tasks, which typically do have preconditions and/or effects; more importantly, ϵ tasks serve a special purpose in maintaining "links" to recursive compound tasks as illustrated in the example below.

Example 12. Consider once again the elevator domain from Section 2.3, which has the following two methods for handling compound task *go-to-bottom* for moving down one floor until the ground floor (floor 0) is reached:

$(\mathit{go\text{-}to\text{-}bottom}, [\![\{(1 : \mathit{move\text{-}down}), (2 : \mathit{go\text{-}to\text{-}bottom})\}, (1 \prec 2) \wedge (\neg \mathit{floor}(0), 1)]\!])$
$(\mathit{go\text{-}to\text{-}bottom}, [\![\{(1 : \epsilon)\}, (\mathit{floor}(0), 1)]\!])$.

[20] Alternatively, the latter can be done by modifying the given task network $[\![s, \phi]\!]$ by adding a conjunct to the constraint formula to take the goal condition $\mathcal{G}$ into account, to obtain the network $[\![s, \phi \wedge \bigwedge_{l \in \mathcal{G}'} (\mathit{last}[n_1, \ldots, n_k], l)]\!]$, where $\{n_1, \ldots, n_k\}$ is the set of task labels occurring in s, and $\mathcal{G}'$ is the set of literals occurring in $\mathcal{G}$.

If the elevator is initially at the second floor, then the decomposition tree for hybrid-plan $[\![\{(0 : \textit{go-to-bottom})\}, \mathit{true}]\!]$ is shown in the figure below. Observe that, given the labelled primitive plan solution $\tau = (1 : \textit{move-down}) \cdot (3 : \textit{move-down}) \cdot (5 : \epsilon)$, the preferred specialisation for the decomposition tree shown is $[\![\{(0 : \textit{go-to-bottom})\}, \mathit{true}]\!]$. This cannot be obtained if the initial cut π does not contain node $(5 : \epsilon)$, because state constraint $(\textit{floor}(0), 5)$ will not hold without this node, resulting in it being impossible to "abstract out" into higher nodes.



■

At any point in time, the algorithm maintains a "current" *cut* π, which is initially a perfect justification. In line 4, a node u in the tree is selected for abstraction: if all children of u are part of the current cut, and the constraints imposed on the children of u are indeed satisfied (line 5), then all the children of u are abstracted into node u (line 6). Therefore, the abstraction process relies not only on the children of a node being present in the current cut, but also on it being possible to satisfy the parent node's constraint formula, with respect to the current cut and state.

It is not difficult to see that the abstraction process is carried out bottom-up, by performing the abstraction of all nodes at a level k before abstracting nodes at level $k + 1$. Thus, given linearisation $\tau' = (2 : a_1) \cdot (9 : a_5) \cdot (10 : a_6) \cdot (12 : a_7) \cdot (13 : a_8) \cdot (15 : a_9)$ for example, and the tree in Figure 7, the hybrid-plan that is eventually computed as a preferred specialisation is $h = [\![s, \phi]\!]$, with

$$s = \{(2 : a_1), (8 : t_4), (11 : t_5), (14 : t_6)\}, \text{ and}$$
$$\phi = (2 \prec 8) \wedge (2 \prec 11) \wedge (8 \prec 14) \wedge (11 \prec 14) \wedge (2 \prec 14).$$

Observe that this is a partially-ordered plan, since the execution of compound tasks with labels 8 and 11 may be interleaved.

Finally, since nodes that decompose into ϵ tasks (e.g. $(16 : t_0)$ in Figure 7) may be abstracted, i.e., added to the current cut π, line 10 removes any such "trivially" abstracted nodes from the final cut. This is because we are interested in finding *minimal* specialisations: those that only include compound tasks that "contribute" to the perfect justification.

The algorithm can be proved correct with respect to Definition 12 (Preferred Specialisation). In fact, it computes not just any preferred specialisation, but a minimal one.

Theorem 4. *Algorithm 1 always terminates and returns a minimal preferred specialisation of hybrid-plan h within $\mathcal{T}_\tau$ for $\mathcal{H}$.*

Proof. Termination (of loops in lines 3 and 4) follows trivially by the fact that the tree (and its height) is finite. To prove that the algorithm returns a preferred specialisation of hybrid-solution h within $\mathcal{T}_\tau$ for $\mathcal{H}$, we claim that any value of cut π, after line 2 and before line 10, conforms to the first two conditions of Definition 12 (Preferred Specialisation). Since the only line where π is modified is line 6, we prove the claim by induction on the number of times k that line 6 is executed.

For the base case, we take $k = 0$. Then, $\pi = \{(n : t) \mid (n : t) \in \tau'\}$ is the "low-level" cut of labelled primitive tasks. Since the projected linearisation $\tau|_{\mathit{actions}(\mathcal{T}|_\pi)}$ is a perfect justification for $\langle \mathcal{I}, \mathcal{G}, \mathit{Op}\rangle$, and since the projected full decomposition tree $\mathcal{T}_\tau|_\pi$—in which all constraint formulas are *true*—is executable in $\mathcal{I}$ relative to *Op*, the first two conditions of Definition 12 hold for π.

Suppose the claim holds for any $k \leq x$, for some $x \in \mathbb{N}_0$. Now we show that the claim also holds for $k = x+1$. Let π_{k-1} be the value of cut π after line 6 is executed $k-1$ times, and let π_k be the value of cut π after line 6 is executed k times. Then, from the algorithm we know that $\pi_k = (\pi_{k-1} \setminus \mathit{children}(u, \mathcal{T})) \cup \{u\}$ for some node u in the decomposition tree, and from the induction hypothesis we know that cut π_{k-1} conforms to the first two conditions of Definition 12. Since the only new task in π_k (relative to π_{k-1}) is u, and $\langle \mathcal{T}_\tau|_{\pi_{k-1}}, \mathcal{I}\rangle \models \ell_V(u)$ holds according to line 5, the first two conditions of Definition 12 are also met for π_k, and our claim holds.

Next, we show that the third and final condition of Definition 12 also holds for any computed cut. Let π be any cut immediately before line 10 in the algorithm, and let S denote the following statement: there does not exist a node $u \in V(\mathcal{T})$, with $u \neq (\mathit{root} : \epsilon)$, that can be "abstracted out", i.e., such that $\mathit{children}(u, \mathcal{T}) \subseteq \pi$ and $\langle \mathcal{T}_\tau|_\pi, \mathcal{I}\rangle \models \ell_V(u)$. Observe from the algorithm that S holds. To show that the third condition of the definition holds for π, it is sufficient to prove that S entails that there does not exist a cut π' in $\mathcal{T}$, with $|\pi'| < |\pi|$, that dominates π in $\mathcal{T}$ and such that $\mathcal{T}_\tau|_{\pi'}$ is executable in $\mathcal{I}$ relative to *Op*.

Let us assume the contrary (i.e., that S holds but that the third condition of the definition does not). Let $\mathcal{T}' = \mathcal{T}|_{\pi'}$ and $\tau' = \tau|_{\mathit{leaves}(\mathcal{T}')}$. Since π' dominates π, they are both (valid) cuts, and π contains *all* tasks (including ϵ tasks) in linearisation τ or their corresponding abstract nodes, it is not difficult to see that there must exist a sequence of cuts

$$(\pi'_1 = \pi') \cdot \pi'_2 \cdot \ldots \cdot (\pi'_k = \pi)$$

such that for each $i \in \{2, \ldots, k\}$, with $k > 1$, we have that $\pi'_i = (\pi'_{i-1} \setminus \{u\}) \cup \mathit{children}(u, \mathcal{T}')$ for some $u \in \pi'_{i-1}$.[21] Therefore, there is a node $u \in V(\mathcal{T}')$, with $u \neq (\mathit{root} : \epsilon)$, such that $\mathit{children}(u, \mathcal{T}') \subseteq \pi$. Moreover, since $\mathcal{T}'_{\tau'}$ is executable in $\mathcal{I}$ relative to *Op*, we know that $\langle \mathcal{T}'_{\tau'}, \mathcal{I}\rangle \models \ell_V(u)$, and therefore, that $\langle \mathcal{T}_\tau|_\pi, \mathcal{I}\rangle \models \ell_V(u)$. Consequently, S cannot hold, which contradicts our assumption. Thus, hybrid-plan $h_\pi = [\![\pi, \Phi[\mathcal{T}_\tau, \pi]]\!]$ is indeed a preferred specialisation of hybrid-solution h within $\mathcal{T}_\tau$ for $\mathcal{H}$.

Finally, the hybrid-plan returned by the algorithm is a minimal preferred specialisation of h within $\mathcal{T}_\tau$ due to line 10, which removes from π labelled tasks that are either labelled ϵ tasks, or those for which only labelled ϵ tasks occur in leaves. □

Note that while we do extract a perfect justification from τ in line 1, which is NP-hard [21], we could instead use any other function that extracts a "desirable" primitive solution from τ. For example, we could extract a so-called *well justification* [21] instead, which is a weaker notion than perfect justification, but computable in polynomial time on the length of the given primitive solution. An action is well justified if and only if it cannot be removed from the plan without violating its correctness. While this means that no individual action in a well justified plan can be unnecessary, the plan may still have groups of actions that are unnecessary, making the plan "redundant" as per the definition of a perfect justification.

The following result states that once a "desirable" primitive plan has been obtained in line 1, Algorithm 1 runs in polynomial time on the size of the decomposition tree $\mathcal{T}$.

Lemma 2. *Algorithm 1, after completion of line 1, runs in polynomial time on the size of the decomposition tree $\mathcal{T}$.*

[21] $k > 1$ because $|\pi'| < |\pi|$

7 Related Work

Perhaps the most closely related work to ours is that of [30], where algorithms are developed for generating an "optimal" hybrid-solution, i.e., one that yields an optimal primitive plan solution, given a cost function for actions, an initial state, a goal condition, and a simplified hierarchical domain as input. Their approach associates compound tasks with models comprising optimistic and pessimistic costs to reach given states, and their algorithms resemble classical planning algorithms but also take into account the compound task models. Their preference for optimal hybrid-solutions is similar to how we prefer hybrid-solutions that yield non-redundant primitive plan solutions, though we additionally require hybrid-solutions to be maximally-abstract. Moreover, unlike [30], we do not focus on an algorithm for generating hybrid-solutions: we assume a hybrid-solution is already provided (using any algorithm) and focus on 'improving' it.

The work of [27] is similar to that of [30]: in both approaches, the planning problem contains a goal condition. The former work describes a "hybrid planning" algorithm, where classical planning is done alongside the HTN decomposition of compound tasks. However, unlike our approach and that of [30], the algorithms in [27] do not produce hybrid-solutions. Nonetheless, we have borrowed some crucial insights and notions from their approach, particularly the importance of respecting user-intent [22, 27] when planning in the context of HTN-like know-how. To this end, they illustrate certain auxiliary notions, including maximal abstractness and dominance, which we concretise and analyse in our work. Unlike their work, ours provides an account of abstraction that also takes into account the notion of redundancy, which we consider relevant when classical planning is in the context of HTN-like available knowledge. The strand of work on HTN planning with "task insertion" (TIHTN) [24, 48, 3, 2] is related to [27] with the main difference being that the planning problem in the latter (and in our work) contains a goal condition, whereas the planning problem in TIHTN planning contains an initial task network.

Another relevant planner that fuses classical and HTN-like planning is GoDel [41]. GoDel is a hierarchical goal-based planner that uses so called "methods"—sequences of subgoals—to achieve classical goal conditions. Importantly, their methods are simply suggestions on how to achieve a goal, as opposed to strict requirements on how to achieve a goal (task) as in standard HTN planning. This allows GoDel to be complete: if there exists a primitive plan for a problem then GoDel will find it, even if the plan cannot be derived via the methods supplied. This is similar to how we compromise on user-intent to avoid redundancy, though they do it to avoid losing completeness. Besides not being an HTN planner *per se*, GoDel differs from our work in that the issues of abstraction and redundancy are not explored, which is the central focus of our work. Thus, our work could be used to inform which plans GoDel should strive to find.

In [3], the HTN, GoDel, and TIHTN approaches are combined into a unified framework, together with a semantics for "task sharing", i.e., merging a pair of unconstrained, identical tasks in a task network by removing one of them and updating constraints accordingly. Task sharing seems to be a special case of removing redundant tasks, which does not require tasks to be unconstrained, and may involve removing unique tasks in the network.

Efforts on adding classical planning to BDI-like systems have focused on synthesising plans that are composed of low-level steps (e.g., [13]) or primitive actions (e.g., [47, 33, 10]). In contrast, we have argued that hybrid-plans are more appropriate in a BDI execution context, as these promote user-intent while retaining the flexibility and robustness properties of such systems. Indeed, hybrid-plans also represent the kinds of plans that are manually written and stored in the agent's plan library. Our previous work [39, 38] also aimed at adding planning into BDI systems, but only HTN-style planning as a local lookahead construct over BDI procedural knowledge. As such, that approach did not deal with the problem of generating novel plans not already available in the agent's plan library.

Finally, there are approaches that focus on using hierarchical domain knowledge to speed up classical planning. In [23, 5] the authors present algorithms that use domain control knowledge inherent in ConGolog/Golog programs as heuristics in classical planning. They do this by modifying planning operators in order to restrict the applicability of their preconditions, and include additional "bookkeeping" operators to guide the search toward actions that are "desirable" as per the control knowledge. Their approach is similar to [4], which embeds HTN domain control knowledge into classical planning problems in order to speed up classical planning. In [8], sequences of primitive actions, called "macro" actions, are learnt automatically from sample primitive solutions to planning problems, and the former are then used to speed up classical planning. Unlike the works described, we do not deal here with guiding a classical planner toward finding a suitable *primitive* plan: we focus instead on defining desirable notions of *hybrid-plans* and on improving a given one, by achieving a balance between abstraction and redundancy.

8 Conclusion and Future Work

In this work, we have set the foundations for classical planning in BDI-like (i.e., hierarchical) agent systems. Specifically, we have argued that plans ought to respect and re-use the user-supplied hierarchical BDI/HTN plan structures while at the same time avoiding the redundancy inherent in resulting solutions. In doing so we have addressed an intrinsic tension between striving for abstraction and avoiding redundancy, by first characterising the set of "ideal" hybrid-plans, which are non-redundant, minimal, and maximally abstract, and then developing a more limited but feasible account of a "preferred" hybrid-plan in which the given hybrid-plan is "specialised" into a new one that is non-redundant but preserves abstraction as much as possible. Our analysis of the properties of the presented notions showed, in particular, that ideal hybrid-plans and preferred specialisations always exist provided the planning problem can be solved, and that any ideal hybrid-plan occurring within a decomposition tree is indeed also a preferred specialisation of the tree. Our final contribution was an algorithm that computes, in polynomial time, a preferred specialisation, given a hybrid-plan and one of its decomposition trees.

There are at least two interesting directions for future work.[22] First, as stated by [32], removing redundant steps is only reasonable in some domains (e.g. in the Mars Rover domain in Figure 1)—doing so is not acceptable in some others, e.g. if the HTN structures embody strong preferences from the user, or necessary restrictions on solutions in order to express certain problems, such as the language intersection problem [19]. To cater for strong preferences we could borrow ideas from [45] and generalise our framework with a more flexible account in which all HTN preferences are assumed to be strong, and a redundant task is only removed if the user has stated separately that the task is not strongly preferred.

It would also be interesting to generalise Algorithm 1 so that the only input is a hybrid planning problem $\mathcal{H} = \langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$. The algorithm would find a primitive solution for the corresponding classical planning problem $\langle \mathcal{I}, \mathcal{G}, Op \rangle$, extract a perfect justification from the solution, and use HTN domain $\mathcal{D}$ in order to abstract the perfect justification. However, the abstraction process is likely to be computationally harder than in Algorithm 1, as multiple

[22] We thank the AI group at the University of Toronto and an anonymous reviewer for these ideas.

abstraction alternatives may need to be tried if the steps to be abstracted occur in multiple methods or method instances.

9 Acknowledgements

The authors are grateful to the two reviewers for their thorough study of the work, and for their insightful and detailed comments which have helped improve this paper significantly. This work was partially supported by Agent Oriented Software and the Australian Research Council (grant LP0882234).

10 Appendices

A Auxiliary Definitions

In this appendix we define notions related to HTN reductions and the evaluation of HTN constraint formulas. To determine whether an HTN constraint formula attached to a node occurring in a full decomposition tree is satisfied, each constraint occurring in the formula needs to be evaluated against the given initial state $\mathcal{I}$ and the full tree's linearisation τ. A constraint is evaluated by determining whether the primitive actions yielded by tasks associated with the labels mentioned in the constraint are in the correct order in τ (in the case of ordering constraints), or whether certain conditions hold for those actions (in the case of state constraints).

Definition 13 (Satisfying a Constraint Formula). Let $\mathcal{T}_\tau$, with $\tau = u_1 \cdot \ldots \cdot u_m$ and $\mathcal{T} = \langle V, E, \ell_V \rangle$, be a full decomposition tree, let $\mathcal{I}$ be a state, and let *Op* be an operator library. Moreover, for any label $n \in \mathbb{N}_0$, let $idx(n)$ denote the set of indices of n's leaf-nodes, i.e., $idx(n) = \{i \mid u = (n : t) \in V(\mathcal{T}),\ u' \in leaves(u, \mathcal{T}),\ i \in \{1, \ldots, m\},\ u' = u_i\}$.

Then, a ground constraint formula ϕ is *satisfied* in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and *Op* if ϕ is satisfied relative to $\mathcal{I}$ and *Op*, where ϕ is evaluated as follows:

1. $(c = c')$ is true if c and c' are the same constant symbols;
2. $first[n, n', \ldots]$ evaluates to $min(idx(n) \cup idx(n') \cup \ldots)$;
3. $last[n, n', \ldots]$ evaluates to $max(idx(n) \cup idx(n') \cup \ldots)$;
4. $(n \prec n')$ is true if $max(idx(n)) < min(idx(n'))$;
5. (l, n) is true if $Res^*(u_1 \cdot \ldots \cdot u_{min(idx(n))-1}, \mathcal{I}, Op) \models l$, i.e., (ground) literal l is true in the state that results from applying to $\mathcal{I}$ the actions in τ up to the action immediately before n;
6. (n, l) is true if $Res^*(u_1 \cdot \ldots \cdot u_{max(idx(n))}, \mathcal{I}, Op) \models l$;
7. (n, l, n') is true if $Res^*(u_1 \cdot \ldots \cdot u_k, \mathcal{I}, Op) \models l$, for all $max(idx(n)) \leq k < min(idx(n'))$;
8. logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic. ■

Below we list two important definitions from [19] for convenience.[23] The first defines a *completion* of a primitive task network, which is basically an ordering and grounding of the primitive tasks in the network such that the ordering conforms with the constraints imposed on those tasks by the network. The second definition concretises the notion of an HTN reduction.

Definition 14 (Completion of a Task Network (adapted from [19])). Let $\sigma = act_1 \cdot \ldots \cdot act_m$ be a plan, *Op* be an operator-library, S_0 be the initial state, and $S_i = Res(act_i, S_{i-1}, Op)$ for $i \in \{1, \ldots, m\}$ be the intermediate states, which are all defined (i.e., the preconditions of each act_i are satisfied in S_{i-1} and thus actions in the plan are executable). Let $d = [\![\{(n_1 : act'_1), \ldots, (n_m : act'_m)\}, \phi]\!]$ be a ground primitive task network, and ρ be a permutation such that whenever $\rho(i) = j, act'_i = act_j$. Then, $\sigma \in comp(d, S_0, \mathcal{D})$ if the constraint formula ϕ of d is satisfied. The constraint formula is evaluated as follows:

1. $(c_i = c_j)$ is true if c_i, c_j are the same constant symbols;
2. $first[n_i, n_j, \ldots]$ evaluates to $min(\{\rho(i), \rho(j), \ldots\})$;
3. $last[n_i, n_j, \ldots]$ evaluates to $max(\{\rho(i), \rho(j), \ldots\})$;
4. $(n_i \prec n_j)$ is true if $\rho(i) < \rho(j)$;
5. (l, n_i) is true if l holds in $S_{\rho(i)-1}$;

[23] We have adapted their notation slightly to be more in line with the notation used in this paper.

6. (n_i, l) is true if l holds in $S_{\rho(i)}$;
7. (n_i, l, n_j) is true if l holds for all S_k, $\rho(i) \leq k < \rho(j)$; and
8. logical connectives $\neg, \wedge, \vee$ are evaluated as in propositional logic.

If d contains compound tasks, then $comp(d, S_0, \mathcal{D}) = \emptyset$, and if d is a primitive task network containing variables, then $comp(d, S_0, \mathcal{D}) = \{\sigma \mid \sigma \in comp(d', S_0, \mathcal{D}),\ d' \text{ is a ground instance of } d\}$. ■

Definition 15 (HTN Reduction [19]). Let $d = [\![\{(n : t), (n_1 : t_1), \ldots, (n_m : t_m)\}, \phi]\!]$ be a task network containing a non-primitive task t. Let $me = (t', [\![\{(n'_1 : t'_1), \ldots, (n'_k : t'_k)\}, \phi']\!])$ be a method,[24] and θ be the most general unifier of t and t'. Then,

$$reduce(d, n, me) = [\![\{(n'_1 : t'_1)\theta, \ldots, (n'_k : t'_k)\theta, (n_1 : t_1)\theta, \ldots, (n_m : t_m)\theta\}, \phi'\theta \wedge \psi]\!],$$

where ψ is obtained from $\phi\theta$ with the following modifications:

- replace $(n \prec n_j)$ with $(\mathit{last}[n'_1, \ldots, n'_k] \prec n_j)$, as n_j must come after every task in the decomposition of n;
- replace $(n_j \prec n)$ with $(n_j \prec \mathit{first}[n'_1, \ldots, n'_k])$;
- replace (l, n) with $(l, \mathit{first}[n'_1, \ldots, n'_k])$, as l must be true immediately before the first task in the decomposition of n;
- replace (n, l) with $(\mathit{last}[n'_1, \ldots, n'_k], l)$, as l must be true immediately after the last task in the decomposition of n;
- replace (n, l, n_j) with $(\mathit{last}[n'_1, \ldots, n'_k], l, n_j)$;
- replace (n_j, l, n) with $(n_j, l, \mathit{first}[n'_1, \ldots, n'_k])$;
- everywhere that n appears in ϕ in a *first*[] or a *last*[] expression, replace it with $n'_1, \ldots, n'_k$. ■

The set of all reductions of task network d relative to HTN domain $\mathcal{D}$ is defined as follows:[25]

$$\mathit{red}(d, \mathcal{D}) = \{d' \mid d' = \mathit{reduce}(d, n, m), (n : t) \in s, m \in \mathit{Me}, d = [\![s, \phi]\!]\}.$$

B Proofs for Lemmas

Proof of Lemma 1 (p. 20). We prove this by induction on the length $k > 0$ of the complete decomposition trace $\lambda = d_1 \cdot \ldots \cdot d_k$. Recall that a complete decomposition trace is ground, and that its last task network mentions only primitive tasks. In what follows, let the induced decomposition tree $\mathcal{T} = \langle V, E, \ell_V \rangle$, and root $rt = (root : \epsilon)$.

[Base Case: $k = 1$.] In this case, trace λ is simply a primitive task network. If $d_k = [\![\emptyset, \mathit{true}]\!]$, then the theorem holds trivially because $comp(d_k, \mathcal{I}, \mathcal{D})$ consists of the empty plan, and the full induced tree $\mathcal{T}_\tau$, where $\mathcal{T} = \langle \{rt\}, \emptyset, \{(rt, \mathit{true})\} \rangle$ and τ is the empty plan, is executable in $\mathcal{I}$ relative to *Op*. If $d_k = [\![s_k, \phi_k]\!]$ such that s_k is non-empty, then there are two cases to consider.
[$\Rightarrow$] From the assumption of the theorem we have that $\sigma = act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$ and $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$. To show that the full decomposition tree $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to *Op* (Definition 10) we rely on definitions 13 (Satisfying a Constraint Formula) and 14 (Completion of a Task Network). For Definition 13, let us take the following vertices $V = \{rt, (n_1 : act_1), \ldots, (n_m : act_m)\}$, the following edges $E = \{(rt, (n_1 : act_1)), \ldots, (rt, (n_m : act_m))\}$, and the following labels $\ell_V = \{(rt, \phi_k), ((n_1 : act_1), \mathit{true}), \ldots, ((n_m : act_m), \mathit{true})\}$ as the induced tree $\mathcal{T}$. For Definition 14, let us take plan σ together with task network $d_k = [\![\{(n'_1 : act'_1), \ldots, (n'_m : act'_m)\}, \phi_k]\!]$. Then, observe from the two definitions that indices $\mathit{idx}(n'_i) = \{\rho(i)\}$ for any $i \in \{1, \ldots, m\}$, i.e., the LHS and RHS both refer to the same index in σ for the given "reference" to an action. Moreover, since d_k is a primitive task network, observe from Definition 14 that $min(\mathit{idx}(n)) = \mathit{idx}(n)$, and likewise, that $max(\mathit{idx}(n)) = \mathit{idx}(n)$, for any primitive task label n. Then, it follows straightforwardly that whenever a condition (numbered from 1 to 8) for evaluating constraint formula ϕ_k holds in Definition 14, the corresponding condition also holds in Definition 13 for formula $\ell_V(rt) = \phi_k$. Therefore, combined with the fact that $\mathit{Res}^*(act_1 \cdot \ldots \cdot act_m, \mathcal{I}, \mathit{Op})$ is defined due to Definition 14 (Completion of a Task Network), we can conclude that $\mathcal{T}_\tau$ is indeed executable

[24] All variables and task labels in the method must be renamed with variables and task labels that do not appear anywhere else.

[25] We have slightly adapted the definition from [19] to remove the mention of a state, which is not necessary.

in $\mathcal{I}$ relative to *Op*. The proof for case $\Leftarrow$ is analogous.

[Induction Hypothesis] Let us assume that the theorem holds if $k \leq x$, for some $x \in \mathbb{N}_1$.

[Inductive Step] Let $k = x + 1$. There are two cases to consider.
[$\Rightarrow$] From the assumption of the theorem we have that $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$. Moreover, from the induction hypothesis, there exists a full decomposition tree $\mathcal{T}'_\tau$, with $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$, that is executable in $\mathcal{I}$ relative to *Op*, such that $\mathcal{T}' = \langle V', E', \ell'_V \rangle$ is the induced decomposition tree of $d_2 \cdot \ldots \cdot d_k$ (recall that trace $\lambda = d_1 \cdot d_2 \cdot \ldots \cdot d_k$), where each $d_i = [\![s_i, \phi_i]\!]$. Finally, we know from Definition 8 (Decomposition Trace) that $d_2 = reduce(d_1, n, me)$ for some task $(n : t) \in s_1$ and ground method $me = (t, [\![s^{me}, \phi^{me}]\!])$, where $s_2 = \Big(s_1 \setminus \{(n : t)\}\Big) \cup s^{me}$, and moreover, we know from Definition 15 (HTN Reduction) that $\phi_2 = \phi'_1 \wedge \phi^{me}$ (where ϕ'_1 is obtained from ϕ_1 by doing the modifications to ϕ_1 as defined in Definition 15). We shall now prove that $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to *Op* by showing that all constraint formulas (labels) of tasks in the induced tree $\mathcal{T}$ are satisfied.

Observe from Definition 9 (Induced Decomposition Tree) that the only differences between induced trees $\mathcal{T}$ and $\mathcal{T}'$ are the following:

(1) the set of vertices $V = V' \cup \{(n : t)\}$, where $(n : t)$ is the task that was reduced;
(2) *children*$((n : t), \mathcal{T}) = s^{me}$;
(3) *children*$(rt, \mathcal{T}) = \Big(\textit{children}(rt, \mathcal{T}') \setminus s^{me}\Big) \cup \{(n : t)\}$; and
(4) the "new" labels $\ell_V = \Big(\ell'_V \setminus \{(rt, \phi_2)\}\Big) \cup \{(rt, \phi_1), \big((n : t), \phi^{me}\big)\}$.

Thus, we need to show that all the "new" constraints occurring in ℓ_V are satisfied in $\mathcal{T}_\tau$ (condition (4) above). However, since we already know from the induction hypothesis that constraint formula $\ell'_V(rt) = \phi_2 = \phi'_1 \wedge \phi^{me}$ (for some formula ϕ'_1) is satisfied in $\mathcal{T}'_\tau$ (relative to $\mathcal{I}$ and *Op*), and therefore that formula $\ell_V((n : t)) = \phi^{me}$ is also satisfied in $\mathcal{T}_\tau$, we only need to show that $\ell_V(rt) = \phi_1$ is satisfied in $\mathcal{T}_\tau$. To this end we consider the possible "structural" differences between ϕ_1 and ϕ'_1. In what follows, let $s^{me} = \{(n_1^{me} : t_1^{me}), \ldots, (n_k^{me} : t_k^{me})\}$.

First, we consider the case where a constraint $(\textit{last}[n_1^{me}, \ldots, n_k^{me}] \prec n')$ occurring in ϕ'_1 has the form $(n \prec n')$ in ϕ_1 (recall n refers to the task that was reduced). By Definition 13 (Satisfying a Constraint Formula), the constraint $(\textit{last}[n_1^{me}, \ldots, n_k^{me}] \prec n')$ evaluates to

$$\max\Big(\bigcup_{j \in \{1,\ldots,k\}} idx(n_j^{me})\Big) < \min(idx(n')), \tag{5}$$

which holds in $\mathcal{T}'_\tau$ due to the induction hypothesis. By the same definition, $(n \prec n')$ evaluates to

$$\max(idx(n)) < \min(idx(n')). \tag{6}$$

Then, since $\{(n_1^{me} : t_1^{me}), \ldots, (n_k^{me} : t_k^{me})\} = \textit{children}((n : t), \mathcal{T})$ (from condition (2) above), it follows from Definition 13 that equations (5) and (6) are equivalent. Therefore, $(n \prec n')$ is indeed satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and *Op*.

Second, we consider the case where a constraint $(\textit{last}[n_1^{me}, \ldots, n_k^{me}, n'_1, \ldots, n'_j] \prec n')$ occurring in ϕ'_1 has the form $(\textit{last}[n, n'_1, \ldots, n'_j] \prec n')$ in ϕ_1. As before, $(\textit{last}[n_1^{me}, \ldots, n_k^{me}, n'_1, \ldots, n'_j] \prec n')$ evaluates to

$$\max\Big(\bigcup_{i \in \{1,\ldots,k\}} idx(n_i^{me}) \cup \bigcup_{i \in \{1,\ldots,j\}} idx(n'_i)\Big) < \min(idx(n')),$$

which holds in $\mathcal{T}'_\tau$ due to the induction hypothesis, and which, as in the previous case, is equivalent to how $(\textit{last}[n, n'_1, \ldots, n'_j] \prec n')$ is evaluated. The remaining cases can be proved similarly: e.g. where a constraint $(n' \prec \textit{last}[n_1^{me}, \ldots, n_k^{me}])$ occurring in ϕ' has the form $(n' \prec n)$ in ϕ. This concludes the proof for the claim that the constraint formula (label) of each node in $\mathcal{T}$ is satisfied in $\mathcal{T}_\tau$ relative to $\mathcal{I}$ and *Op*. Combined with the fact that *Res*$^*(act_1 \cdot \ldots \cdot act_m, \mathcal{I}, Op)$ is defined due to Definition 14 (Completion of a Task Network), we can conclude that $\mathcal{T}_\tau$ is indeed executable in $\mathcal{I}$ relative to *Op*.

[$\Leftarrow$] For this second case of the induction hypothesis, we have from the assumption of the theorem that $\mathcal{T}_\tau$ is executable in $\mathcal{I}$ relative to *Op*, where $\tau = (n_1 : act_1) \cdot \ldots \cdot (n_m : act_m)$, and we need to prove that $act_1 \cdot \ldots \cdot act_m \in comp(d_k, \mathcal{I}, \mathcal{D})$. The proof for this case is analogous to the proof for case $\Rightarrow$. $\square$

Proof of Lemma 2 (p. 27). The only non-trivial lines in the algorithm are 5 and 11. Line 11 runs in polynomial time because $\Phi[\mathcal{T}_\tau, \pi]$ can be computed by first finding all 2-permutations of set π using a nested *for* loop, and then determining for each pair $(n_1 : t_1), (n_2 : t_2)$ whether all leaves of n_1 in $\mathcal{T}$ occur before all leaves of n_2 in $\mathcal{T}$, with respect to τ. Line 5 runs in polynomial time for the following reason. Observe from Definition 13 (Satisfying a Constraint Formula) that a constraint formula is evaluated by first assigning one truth value (*true* or *false*) to each ground conjunct (constraint), and then evaluating the resulting interpretation, of which the latter has a polynomial time algorithm [9]. When assigning truth values to constraints, the only non-trivial part is computing the state that results from applying a labelled primitive plan τ' to a state $\mathcal{I}$, i.e., $\mathit{Res}^*(\tau', \mathcal{I}, \mathit{Op})$. Recall from Section 2.1 that for each action in τ', this simply involves first checking whether the atom associated with each literal in the precondition of the action is a member of a given state, and then adding/removing two sets of ground atoms (i.e., the action's add and delete lists) to/from the state.□

C Graphs and Trees

This appendix defines some notions involving graphs. First, we define a *directed graph* and related terms.

Definition 16 (Graph Terminology). A *directed graph* G is the tuple $\langle V(G), E(G)\rangle$, where $V(G)$ is a set of vertices and $E(G) \subseteq V(G) \times V(G)$ is a set of edges. For any edge $(v_1, v_2) \in E(G)$, we call v_1 the *parent* vertex and v_2 the *child* vertex. Given a directed graph G, we define the following additional terms.

- G is *cyclic* if there exists a sequence of vertices $v_1 \cdot v_2 \cdot \ldots \cdot v_n \in V^n$, such that
  - $\forall i \in \{1, \ldots, n-1\}$, $(v_i, v_{i+1}) \in E$, i.e., for each pair of adjacent vertices in the sequence, there is an edge directed from the left vertex of the pair to its right vertex; and
  - $(v_n, v_1) \in E$, i.e., there is an edge directed from the last vertex in the sequence to the first.
- G is *rooted* if $|\{v \mid v \in V, \forall v' {\in} V\ ((v', v) \notin E)\}| = 1$, i.e., there is exactly one vertex without a parent vertex. Given a rooted, directed graph $G' = \langle V', E'\rangle$, the *root* of G', denoted $\mathit{root}(G')$, is the vertex $v \in V'$ such that for each $v' {\in} V'$, $(v', v) \notin E'$. ■

A *tree* is a rooted directed graph that is not cyclic. Below we define some relevant terms involving trees.

Definition 17 (Tree Terminology). Let $G = \langle V, E\rangle$ be a tree.

- The *children* of a vertex $v \in V$ in G, denoted $\mathit{children}(v, G)$, is the set $\{v' \mid (v, v') \in E\}$.
- The *descendants* of a vertex $v \in V$ in G, denoted $\mathit{descendants}(v, G)$, is defined inductively as

$$\mathit{descendants}(v, G) = \mathit{children}(v, G) \cup \bigcup_{v' \in \mathit{children}(v,G)} \mathit{descendants}(v', G).$$

- The *leaves* of G, denoted $\mathit{leaves}(G)$, is the set $\{v \mid v \in V,\ (v, v') \notin E\}$. Moreover, the *leaves* of a vertex $v \in V$ in G, denoted $\mathit{leaves}(v, G)$, is the set $(\mathit{descendants}(v, G) \cup \{v\}) \cap \mathit{leaves}(G)$.
- The *height* of a *vertex* $v \in V$ in tree G, denoted $\mathit{height}(v, G)$, is defined as follows. If $v \in \mathit{leaves}(G)$, then $\mathit{height}(v, G) = 0$. Otherwise, let $v_1 \cdot \ldots \cdot v_n$, with $n > 1$, be the longest sequence such that $v = v_1$ and $(v_i, v_{i+1}) \in E$ for all $i \in \{1, \ldots, n-1\}$. Then, $\mathit{height}(v, G) = n - 1$. Finally, the *height* of the tree, denoted $\mathit{height}(G)$, is $\mathit{height}(\mathit{root}(G), G)$. ■

The above definitions trivially generalise to a *vertex-labelled tree*, defined as follows.

Definition 18 (Vertex-Labelled Tree). Let L_V be a finite set of labels. A *vertex-labelled tree* is the tuple $\langle V, E, \ell_V\rangle$, where $\langle V, E\rangle$ is a tree, and $\ell_V : V \mapsto L_V$ is a function that assigns each vertex with a label. Given a vertex $v \in V$, we say that $\ell_V(v)$ is the *label* of v. ■

Finally, we define the notion of *projecting* on a cut in a decomposition tree.

Definition 19 (Projection). The decomposition tree obtained by *projecting* on a cut $\pi \subseteq V(\mathcal{T})$ in a decomposition tree $\mathcal{T} = \langle V, E, \ell_V\rangle$, denoted by $\mathcal{T}|_\pi$, is defined as follows:

$$\begin{aligned}
&\mathcal{T}|_\pi = \langle V', E', \ell'_V\rangle, \text{ where}\\
&\quad V' = \{rt\} \cup \pi \cup \{u' \mid u' \in \mathit{descendants}(u, \mathcal{T}),\ u \in \pi\}, \text{ with } rt = (\mathit{root} : \epsilon);\\
&\quad E' = \{(rt, u) \mid u \in \pi\} \cup \{(u, u') \mid (u, u') \in E \text{ and } u, u' \in V'\}; \text{and}\\
&\quad \ell'_V = \{(rt, \mathit{true})\} \cup \{(u, \phi) \mid (u, \phi) \in \ell_V \text{ and } u \in V',\ u \neq rt\}.
\end{aligned}$$ ■

The notion of projecting on a cut π generalises to a full decomposition tree $\mathcal{T}_\tau$, denoted by $\mathcal{T}_\tau|_\pi$, as follows: $\mathcal{T}_\tau|_\pi = \mathcal{T}'_{\tau'}$ with $\mathcal{T}' = \mathcal{T}|_\pi$ and $\tau' = \tau|_{\mathit{leaves}(\mathcal{T}')}$, where for any set of labelled tasks π and sequence of labelled primitive tasks τ, projection $\tau|_\pi$ is the largest subsequence τ' of τ such that for each task $u \in \tau'$, $u \in \pi$.

References

1. Alami R, Chatila R, Fleury S, Ghallab M, Ingrand F (1998) An architecture for autonomy. The International Journal of Robotics Research (IJRR) 17(4):315–337
2. Alford R, Bercher P, Aha DW (2015) Tight bounds for HTN planning with task insertion. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1502–1508
3. Alford R, Shivashankar V, Roberts M, Frank J, Aha DW (2016) Hierarchical planning: Relating task and goal decomposition with task sharing. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 3022–3029
4. Alford RW, Kuter U, Nau D (2009) Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1629–1634
5. Baier JA, Fritz C, McIlraith SA (2007) Exploiting procedural domain control knowledge in state-of-the-art planners. In: Proc. of the International Conference on Automated Planning and Scheduling (ICAPS), pp 26–33
6. Benfield SS, Hendrickson J, Galanti D (2006) Making a strong business case for multiagent technology. In: Proc. of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), pp 10–15
7. Bordini RH, Braubach L, Dastani M, Seghrouchni AEF, Gómez Sanz JJ, Leite Ja, O'Hare GMP, Pokahr A, Ricci A (2006) A survey of programming languages and platforms for Multi-Agent systems. Informatica (Slovenia) 30(1):33–44
8. Botea A, Enzenberger M, Müller M, Schaeffer J (2005) Macro-FF: Improving AI planning with automatically learned macro-operators. Journal of Artificial Intelligence Research (JAIR) 24:581–621
9. Buss SR (1987) The boolean formula value problem is in ALOGTIME. In: Proc. of the ACM Symposium on Theory of Computing (STOC), pp 123–131
10. Claßen J, Eyerich P, Lakemeyer G, Nebel B (2007) Towards an integration of Golog and planning. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1846–1851
11. de Boer FS, Hindriks KV, van der Hoek W, Meyer JJ (2007) A verification framework for agent programming with declarative goals. Journal of Applied Logic 5(2):277–302
12. de Silva L, Padgham L (2004) A comparison of BDI based real-time reasoning and HTN based planning. In: Proc. of the Australian Joint Conference on AI (AI), pp 1167–1173
13. Despouys O, Ingrand FF (1999) Propice-Plan: Toward a unified framework for planning and execution. In: Proc. of the European Conference on Planning (ECP), pp 278–293
14. Dix J, Muñoz-Avila H, Nau DS, Zhang L (2003) IMPACTing SHOP: Putting an AI planner into a multi-agent environment. Annals of Mathematics and Artificial Intelligence 37(4):381–407
15. Dvorak F, Barták R, Bit-Monnot A, Ingrand F, Ghallab M (2014) Planning and acting with temporal and hierarchical decomposition models. In: Proc. of the International Conference on Tools with Artificial Intelligence (ICTAI), pp 115–121
16. Erol K, Hendler J, Nau DS (1994) HTN planning: Complexity and expressivity. In: Proc. of the National Conference on Artificial Intelligence (AAAI), pp 1123–1128
17. Erol K, Hendler J, Nau DS (1994) Semantics for hierarchical task-network planning. Tech. Rep. UMIACS-TR-94-31, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, U.S.A.
18. Erol K, Nau DS, Subrahmanian VS (1995) Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Artificial Intelligence

(AIJ) 76(1-2):75–88
19. Erol K, Hendler JA, Nau DS (1996) Complexity results for HTN planning. Annals of Mathematics and Artificial Intelligence 18(1):69–93
20. Fikes RE, Nilsson NJ (1971) STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence (AIJ) 2(3-4):189–208
21. Fink E, Yang Q (1992) Formalizing plan justifications. In: Proc. of the Conference of the Canadian Society for Computational Studies of Intelligence, pp 9–14
22. Fox M (1997) Natural hierarchical planning using operator decomposition. In: Proc. of the European Conference on Planning (ECP), pp 195–207
23. Fritz C, Baier JA, McIlraith SA (2008) ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for planning and beyond. In: Proc. of the International Conference on Principles of Knowledge Representation and Reasoning (KR), pp 600–610
24. Geier T, Bercher P (2011) On the decidability of HTN planning with task insertion. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1955–1961
25. Georgeff MP, Ingrand FF (1989) Decision making in an embedded reasoning system. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 972–978
26. Ghallab M, Nau DS, Traverso P (2004) Automated Planning: Theory and Practice. Morgan Kaufmann Publishers Inc.
27. Kambhampati S, Mali AD, Srivastava B (1998) Hybrid planning for partially hierarchical domains. In: Proc. of the National Conference on Artificial Intelligence (AAAI), pp 882–888
28. Ljungberg M, Lucas A (1992) The OASIS air-traffic management system. In: Proc. of the Pacific Rim International Conference on Artificial Intelligence (PRICAI), pp 15–18
29. Lloyd JW (1987) Foundations of Logic Programming; (2nd extended ed.). Springer-Verlag
30. Marthi B, Russell SJ, Wolfe JA (2008) Angelic hierarchical planning: Optimal and online algorithms. In: Proc. of the International Conference on Automated Planning and Scheduling (ICAPS), pp 222–231
31. Meneguzzi F, Luck M (2013) Declarative planning in procedural agent architectures. Expert Systems with Applications 40(16):6508–6520
32. Meneguzzi F, de Silva L (2015) Planning in BDI agents: A survey of the integration of planning algorithms and agent reasoning. Knowledge Engineering Review 30(1):1–44
33. Meneguzzi F, Zorzo AF, da Costa Móra M (2004) Propositional planning in BDI agents. In: Proc. of the ACM Symposium on Applied Computing, pp 58–63
34. Minton S, Bresina J, Drummond M (1994) Total-order and partial-order planning: A comparative analysis. Journal of Artificial Intelligence Research (JAIR) 2:227–262
35. Nau DS, Cao Y, Lotem A, Muñoz-Avila H (1999) SHOP: Simple hierarchical ordered planner. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 968–973
36. Nau DS, Au TC, Ilghami O, Kuter U, Murdock JW, Wu D, Yaman F (2003) SHOP2: An HTN planning system. Journal of Artificial Intelligence Research (JAIR) 20:379–404
37. Rao AS (1996) AgentSpeak(L): BDI agents speak out in a logical computable language. In: Proc. of the European workshop on Modelling Autonomous Agents in a Multi-Agent World : agents breaking away (MAAMAW), Springer, pp 42–55
38. Sardina S, Padgham L (2011) A BDI agent programming language with failure recovery, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems (JAAMAS) 23(1):18–70

39. Sardina S, de Silva L, Padgham L (2006) Hierarchical planning in BDI agent programming languages: A formal approach. In: Proc. of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), pp 1001–1008
40. Schattenberg B (2009) Hybrid planning and scheduling. PhD thesis, University of Ulm, Germany
41. Shivashankar V, Alford R, Kuter U, Nau D (2013) The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 2380–2386
42. de Silva L, Sardina S, Padgham L (2009) First Principles Planning in BDI systems. In: Proc. of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), pp 1105–1112
43. de Silva L, Lallement R, Alami R (2015) The HATP hierarchical planner: Formalisation and an initial study of its usability and practicality. In: Proc. of the International Conference on Intelligent Robots and Systems (IROS), pp 6465–6472
44. de Silva L, Sardina S, Padgham L (2016) Summary information for reasoning about hierarchical plans. In: Proc. of the European Conference on Artificial Intelligence (ECAI), pp 1300–1308
45. Sohrabi S, Baier JA, McIlraith SA (2009) HTN planning with preferences. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 1790–1797
46. Wallis P, Rönnquist R, Jarvis D, Lucas A (2002) The automated wingman - using JACK Intelligent Agents for unmanned autonomous vehicles. In: Proc. of the IEEE Aerospace Conference, pp 2615–2622
47. Wilkins DE, Myers KL, Lowrance JD, Wesley LP (1995) Planning and reacting in uncertain and dynamic environments. Journal of Experimental and Theoretical Artificial Intelligence (JETAI) 7(1):197–227
48. Xiao Z, Herzig A, Perrussel L, Wan H, Su X (2017) Hierarchical task network planning with task insertion and state constraints. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI), pp 4463–4469