HEART: Using Abstract Plans As A Guarantee Of Downward Refinement In Decompositional Planning *

Antoine Gréa Laetitia Matignon Samir Aknine

Keywords: Hierarchical planning, HTN, Partial Order Causal Link, POCL, Partial Order Planning, POP, Planning Algorithms

Abstract: In recent years the ubiquity of artificial intelligence raised concerns among the uninitiated. The misunderstanding is further increased since most advances do not have explainable results. For automated planning, the research often targets speed, quality, or expressivity. Most existing solutions focus on one criteria while not addressing the others. However, human-related applications require a complex combination of all those criteria at different levels. We present a new method to compromise on these aspects while staying explainable. We aim to leave the range of potential applications as wide as possible but our main targets are human intent recognition and assistive robotics. We propose the HEART planner, a real-time decompositional planner based on a hierarchical version of Partial Order Causal Link (POCL). It cyclically explores the plan space while making sure that intermediary *high level plans* are valid and will return them as approximate solutions when interrupted. These plans are proven to be a guarantee of solvability. This paper aims to evaluate that process and its results compared to classical approaches in terms of efficiency and quality.

Introduction

Since the early days of automated planning, a wide variety of approaches have been considered to solve diverse types of problems. They all range in expressivity, speed, and reliability but often aim to excel in one of these domains. This leads to a polarization of the solutions toward more specialized methods to tackle each problem. All of these approaches have been compared and discussed extensively in the books of Ghallab *et al.* (2004, 2016).

Partially ordered approaches are popular for their least commitment aspect, flexibility and ability to modify plans to use refinement operations (Weld, 1994). These approaches are often used in applications in robotics and multi-agent planning (Dvorak et al., 2014; Lemai and Ingrand, 2004). One of the most flexible partially ordered approaches is called *Partial Order Causal Link planning (POCL)* (Young and Moore, 1994). It works by refining partial plans consisting of steps and causal links into a solution by solving all flaws compromising the validity of the plan.

Another approach is *Hierarchical Task Networks (HTN)* (Sacerdoti, 1974) that is meant to tackle the problem using composite actions in order to define hierarchical tasks within the plan. Hierarchical domains are considered easier to conceive and maintain by experts mainly because they seem closer to the way we think about these problems (Sacerdoti, 1975).

In our work, we aim to combine HTN planning and POCL planning in such a manner as to generate intermediary high-level plans during the planning process. Combining these two approaches is not new (Biundo and Schattenberg, 2001; Kambhampati et al., 1998; Young and Moore, 1994). Our work is based on *Hierarchical Partial Order Planning (HiPOP)* by Bechon *et al.* (2014). The idea is to expand the classical POCL algorithm with new flaws in order to make it compatible with HTN problems and allowing the production of abstract plans. To do so, we present an upgraded planning framework that aims to simplify and factorize all notions to their minimal forms. We also propose some domain compilation techniques to reduce the work of the expert conceiving the domain.

In all these works, only the final solution to the input problem is considered. That is a good approach to classical planning except when no solutions can be found (or when none exists). Our work focuses on the case when the solution could not be found in time or when high-level explanations are preferable to the complete implementation detail of the plan. This is done by focusing the planning effort toward finding intermediary abstract plans along the path to the complete solution.

In the rest of the paper, we detail how the HiErarchical Abstraction for Real-Time (HEART) planner creates abstract intermediary plans that can be used for various

* Univ Lyon, Université Lyon 1, CNRS, LIRIS, UMR5205, F-69621, LYON, France (first.lastname@liris.cnrs.fr)

applications. First, we discuss the motivations and related works to detail the choices behind our design process. Then we present the way we modeled our own planning framework fitting our needs and then we explain our method and prove its properties to finally discuss the experimental results.

1 Motivations and Potential Applications

Several reasons can cause a problem to be unsolvable. The most obvious case is that no solution exists that meets the requirements of the problem. This has already been addressed by Göbelbecker *et al.* (2010) where "excuses" are being investigated as potential explanations for when a problem has no solution.

Our approach deals with problems that are too difficult to solve within tight time constraints. For example, in robotics, systems often need to be run within refresh rates of several Hertz giving the process only fractions of a second to give an updated result. Since planning is at least EXPSPACE-hard for HTN using complex representation (Erol et al., 1994), computing only the first plan level of a hierarchical domain is much easier in relation to the complete problem.

While abstract plans are not complete solutions, they still display a useful set of properties for various applications. The most immediate application is for explainable planning (Fox et al., 2017; Seegebarth et al., 2012). Indeed a high-level plan is more concise and does not contain unnecessary implementation details that would confuse a non-expert. Recent works focus on matching the domain to a separate human model (Sreedharan, Chakraborti, et al., 2018; Sreedharan, Srivastava, et al., 2018). This requires the creation and maintenance of expansive human domains that are mostly used as dictionaries to explain technical details. Our method will give coherent high level plans that are more concise and simpler than any classical plans.

Another potential application for such plans is relative to domains that work with approximative data. Our main example here is intent recognition which is the original motivation for this work. Planners are not meant to solve intent recognition problems. However, several works extended what is called in psychology the *theory of mind*. That theory is the equivalent of asking "*what would **I** do if I was them ?*" when observing the behavior of other agents. This leads to new ways to use *inverted planning* as an inference tool. One of the first to propose that idea was Baker *et al.* (2007) that use Bayesian planning to infer intentions. Ramirez and Geffner (2009) found an elegant way to transform a plan recognition problem into classical planning. This is done simply by encoding temporal constraints in the planning domain in a similar way as Baioletti *et al.* (1998) describe it to match the observed action sequence. A cost comparison will then give a probability of the goal to be pursued given the observations. Chen *et al.* (2013) extended this with multi-goal recognition. A new method, proposed by Sohrabi *et al.* (2016), makes the recognition fluent centric. It assigns costs to missing or noisy observed fluents, which allows finer details and less preprocessing work than action-based recognition. This method also uses a meta-goal that combines each possible goal and is realized when at least one of these goals is satisfied. Sohrabi *et al.* state that the quality of the recognition is directly linked to the quality and domain coverage of the generated plans. Thus guided diverse planning[2] was preferred along with the ability to infer several probable goals at once.

2 Related Works

HTN is often combined with classical approaches since it allows for a more natural expression of domains making expert knowledge easier to encode. These kinds of planners are named **decompositional planners** when no initial plan is provided (Fox, 1997). Most of the time the integration of HTN simply consists in calling another algorithm when introducing a composite operator during the planning process. The DUET planner by Gerevini *et al.* (2008) does so by calling an instance of a HTN planner based on task insertion called SHOP2 (Nau et al., 2003) to decompose composite actions. Some planners take the integration further by making the decomposition of composite actions into a special step in their refinement process. Such works include the discourse generation oriented DPOCL (Young and Moore, 1994) and the work of Kambhampati *et al.* (1998) generalizing the practice for decompositional planners.

In our case, we chose a class of hierarchical planners based on Plan-Space Planning (PSP) algorithms (Bechon et al., 2014; Bercher et al., 2014; Dvorak et al., 2014) as a reference approach. The main difference here is that the decomposition is integrated into the classical POCL algorithm by only adding new types of flaws. This allows keeping all the flexibility and properties of POCL while adding the expressivity and abstraction capabilities of HTN. We also made an improved planning framework based on the one used by HiPOP to reduce further the

[2]Diverse planning aims to find a set of m plans that are distant of d from one another.

number of changes needed to handle composite actions and to increase the efficiency of the resulting implementation.

As stated previously, our goal is to obtain intermediary abstract plans and to evaluate their properties. Another work has already been done on another aspect of those types of plans. The Angelic algorithm by Marthi *et al.* (2007) exploited such plans in the planning process itself and used them as a heuristic guide. They also proved that, for a given fluent semantics, it is guaranteed that such abstract solutions can be refined into actual solutions. However, the Angelic planner does not address the inherent properties of such abstract plans as approximate solutions and uses a more restrictive totally ordered framework.

3 Definitions

In order to make the notations used in the paper more understandable, we gathered them in table 1. For domain and problem representation, we use a custom knowledge description language that is inspired from RDF Turtle (Beckett and Berners-Lee, 2011) and is based on triples and propositional logic. In that language quantifiers are used to quantify variables `*(x)` (forall x) but can also be simplified with an implicit form: `lost(~)` meaning _" nothing is lost". *For reference, the* exclusive quantifier *we introduced (noted ~) is used for the negation (e.g.* `~(lost(_))` *for* " something is not lost"_) as well as the symbol for nil. All symbols are defined as they are first used. If a symbol is used as a parameter and is referenced again in the same statement, it becomes a variable.

3.1 Domain

The domain specifies the allowed operators that can be used to plan and all the fluents they use as preconditions and effects.

Definition 1 (Domain)**.** A domain is a triplet $\mathcal{D} = \langle E_{\mathcal{D}}, R, A_{\mathcal{D}} \rangle$ where:

- $E_{\mathcal{D}}$ is the set of **domain entities**.
- R is the set of **relations** over $E_{\mathcal{D}}^n$. These relations are akin to n-ary predicates in first order logic.
- $A_{\mathcal{D}}$ is the set of **operators** which are fully lifted *actions*.

Example: The example domain in listing 1 is inspired from the kitchen domain of Ramirez and Geffner (2010).

Table 1: Our notations are adapted from Ghallab *et al.* (2004). The symbol ± shows when the notation has signed variants.

Symbol	Description
$\mathcal{D}, \mathcal{P}$	Planning domain and problem.
$pre(a)$, $eff(a)$	Preconditions and effects of the action a.
$methods(a)$	Methods of the action a.
$\phi^{\pm}(l)$	Signed incidence function : ϕ^- gives the source and ϕ^+ the target step of l. No sign gives a pair corresponding to link l.
$L^{\pm}(a)$	Set of incoming (L^-) and outgoing (L^+) links of step a. No sign gives all adjacent links.
$a_s \xrightarrow{c} a_t$	Link with source a_s, target a_t and cause c.
$causes(l)$	Gives the causes of a causal link l.
$a_a \succ a_s$	A step a_a is anterior to the step a_s.
A_x^n	Proper actions set of x down n levels.
$lv(x)$	Abstraction level of the entity x.
$a \rhd^{\pm} a'$	Transpose the links of action a onto a'.
$l \downarrow a$	Link l partially supports step a.
$\pi \Downarrow a$	Plan π fully supports a.
$\ddagger_f a$	Subgoal: fluent f not supported in step a.
$a_b \otimes l$	Threat: action a_b threatens causal link l.
$a \oplus^m$	Decomposition of action a using method m.
$var : exp$	The colon is to be read as "such that".
$[exp]$	Iverson's brackets: 0 if exp is false, 1 otherwise.

```
take(item) pre (taken(~), ?(item));
    //?(item) is used to make item
    into a variable.
take(item) eff (taken(item));
heat(thing) pre (~(hot(thing)),
    taken(thing));
heat(thing) eff (hot(thing));
pour(thing, into) pre (thing ~(in)
    into, taken(thing));
pour(thing, into) eff (thing in
    into);
put(utensil) pre
    (~(placed(utensil)),
    taken(utensil));
put(utensil) eff (placed(utensil),
    ~(taken(utensil)));

infuse :: Action; //to prevent
    categorization of infuse as a
    variable.
infuse(extract,liquid,container)
    method (
  init(
    infuse(extract,liquid,container)
    ) -> take(extract),
  init(
    infuse(extract,liquid,container)
    ) -> take(liquid),
  take(liquid) -> heat(liquid),
  heat(liquid) -> pour(liquid,
    container),
  take(extract) -> pour(extract,
    container),
```

```
17   pour(liquid, container) -> goal(
       infuse(extract,liquid,container)
       ),
18   pour(extract, container) -> goal(
       infuse(extract,liquid,container)
       ),
19   heat(liquid) -> goal(
       infuse(extract,liquid,container)
       )
20 );
21
22 make :: Action;
23 make(drink) method (
24   init( make(drink) ) ->
       take(spoon),
25   take(spoon) -> put(spoon),
26   init(make(drink)) ->
       infuse(drink,water,cup),
27   infuse(drink,water,cup) ->
       take(cup),
28   take(cup) -> put(cup),
29   put(spoon) -> goal( make(drink) ),
30   infuse(drink,water,cup) -> goal(
       make(drink) ),
31   put(cup) -> goal( make(drink) )
32 );
```

Listing 1: Domain file used in our planner.

Definition 2 (Fluent)**.** A fluent f is a parameterized statement $r(arg_1, arg_2, ..., arg_n)$ where:

- $r \in R$ is a relation/function holding a property of the world.
- $arg_{i \in [1,n]} \in E_{\mathcal{D}}$ are the arguments (possibly quantified).
- $n = |r|$ is the arity of r.

Fluents are signed. Negative fluents are noted $\neg f$ and behave as a logical complement. The quantifiers are affected by the sign of the fluents. We do not use the closed world hypothesis: fluents are only satisfied when another compatible fluent is provided. Sets of fluents have a boolean value that equals the conjunction of all its fluents.

Example: To describe an item not being held, we use the fluent $\neg taken(item)$. If the cup contains water, $in(water, cup)$ is true.

3.2 Plan and hierarchical representation

Definition 3 (Partial Plan / Method)**.** A partially ordered plan is an *acyclic* directed graph $\pi = (S, L)$, with:

- S the set of **steps** of the plan as vertices. A step is an action belonging in the plan. S must contain an initial step I_π and goal step G_π.
- L the set of **causal links** of the plan as edges. We note $l = a_s \xrightarrow{c} a_t$ the link between its source a_s and its target a_t caused by the set of fluents c. If $c = \varnothing$ then the link is used as an ordering constraint.

In our framework, *ordering constraints* are defined as the transitive cover of causal links over the set of steps. We note ordering constraints: $a_a \succ a_s$, with a_a being *anterior* to its *successor* a_s. Ordering constraints cannot form cycles, meaning that the steps must be different and that the successor cannot also be anterior to its anterior steps: $a_a \neq a_s \wedge a_s \not\succ a_a$. In all plans, the initial and goal steps have their order guaranteed: $I_\pi \succ G\pi \wedge \nexists a_x \in S_\pi : a_x \succ I_\pi \vee G_\pi \succ a_x$. If we need to enforce order, we simply add a link without specifying a cause. The use of graphs and implicit order constraints help to simplify the model while maintaining its properties.

The central notion of planning is operators. Instantiated operators are usually called *actions*. In our framework, actions can be partially instantiated. We use the term action for both lifted and grounded operators.

Definition 4 (Action)**.** An action is a parametrized tuple $a(args) = \langle name, pre, eff, methods \rangle$ where:

- *name* is the **name** of the action.
- *pre* and *eff* are sets of fluents that are respectively the **preconditions and the effects** of the action.
- *methods* is a set of **methods** (*partial order plans*) that decompose the action into smaller ones. Methods, and the methods of their enclosed actions, cannot contain the parent action.

Example: The precondition of the operator $take(item)$ is simply a single negative fluent noted $\neg taken(item)$ ensuring the variable $item$ is not already taken.

Composite actions are represented using methods. An action without methods is called *atomic*. It is of interest to note the divergence with classical HTN representation here since normally composite actions do not have preconditions nor effects. In our case, we insert them into abstract plans.

3.3 Input control

In order to verify the input of the domain, the causes of the causal links in the methods are optional. If omitted, the causes are inferred by unifying the preconditions and effects with the same mechanism as in the subgoal resolution in our POCL algorithm. Since we want to guarantee the validity of abstract plans, we need to ensure that user

provided plans are solvable. We use the following formula to compute the final preconditions and effects of any composite action a: $pre(a) = \bigcup_{a_s \in L^+(a)} causes(a_s)$ and $\mathit{eff}(a) = \bigcup_{a_s \in L^-(a)} causes(a_s)$. An instance of the classical POCL algorithm is then run on the problem $\mathcal{P}_a = \langle \mathcal{D}, C_\mathcal{P}, a \rangle$ to ensure its coherence. The domain compilation fails if POCL cannot be completed. Since our decomposition hierarchy is acyclic ($a \notin A_a$, see definition 10) nested methods cannot contain their parent action as a step.

3.4 Problem

Problem instances are often most simply described by two components: the initial state and the goal.

Definition 5 (Problem)**.** The planning problem is defined as a tuple $\mathcal{P} = \langle \mathcal{D}, C_\mathcal{P}, a_0 \rangle$ where:

- $\mathcal{D}$ is a planning domain.
- $C_\mathcal{P}$ is the set of **problem constants** disjoint from the domain constants.
- a_0 is the **root operator** of the problem which methods are potential solutions of the problem.

Example: We use a simple problem for our example domain. The initial state provides that nothing is ready, taken or hot and all containers are empty (all using quantifiers). The goal is to have tea made. For reference, listing 2 contains the problem instance we use as an example.

```
1 init eff (hot(~), taken(~),
      placed(~), ~ in ~);
2 goal pre (hot(water), tea in cup,
      water in cup, placed(spoon),
      placed(cup));
```

Listing 2: Example of a problem instance for the kitchen domain.

The root operator is initialized to $a_0 = \langle \text{""}, s_0, s^*, \{\pi_{lv(a_0)}\} \rangle$, with s_0 being the initial state and s^* the goal specification. The method $\pi_{lv(a_0)}$ is a partial order plan with the initial and goal steps linked together via a_0. The initial partial order plan is $\pi_{lv(a_0)} = (\{I, G\}, \{I \xrightarrow{s_0} a_0 \xrightarrow{s^*} G\})$, with $I = \langle \text{"}init\text{"}, \emptyset, s_0, \emptyset \rangle$ and $G = \langle \text{"}goal\text{"}, s^*, \emptyset, \emptyset \rangle$.

3.5 Partial Order Causal Links

Our method is based on the classical POCL algorithm. It works by refining a partial plan into a solution by recursively removing all of its flaws.

Definition 6 (Flaws)**.** Flaws have a *proper fluent* f and a causing step often called the *needer* a_n. Flaws in a partial plan are either:

- **Subgoals**, *open conditions* that are yet to be supported by another step a_n often called *provider*. We note subgoals $\ddagger_f a_n$.
- **Threats**, caused by steps that can break a causal link with their effects. They are called *breakers* of the threatened link. A step a_b threatens a causal link $l_t = a_p \xrightarrow{f} a_n$ if and only if $\neg f \in \mathit{eff}(a_b) \wedge a_b \not\succ a_p \wedge a_n \not\succ a_b$. Said otherwise, the breaker can cancel an effect of a providing step a_p, before it gets used by its needer a_n. We note threats $a_b \otimes l_t$.

Example: Our initial plan contains two unsupported subgoals: one to make the tea ready and another to put sugar in it. In this case, the needer is the goal step and the proper fluents are each of its preconditions.

These flaws need to be fixed in order for the plan to be valid. In POCL it is done by finding their resolvers.

Definition 7 (Resolvers)**.** Classical resolvers are additional causal links that aim to fix a flaw.

- *For subgoals*, the resolvers are a set of potential causal links containing the proper fluent f in their causes while taking the needer step a_n as their target and a **provider** step a_p as their source.
- *For threats*, we usually consider only two resolvers: **demotion** ($a_b \succ a_p$) and **promotion** ($a_n \succ a_b$) of the breaker relative to the threatened link. We call the added causeless causal link a **guarding** link.

Example: The subgoal for $in(water, cup)$, in our example, can be solved by using the action $pour(water, cup)$ as the source of a causal link carrying the proper fluent as its only cause.

The application of a resolver does not necessarily mean progress. It can have consequences that may require reverting its application in order to respect the backtracking of the POCL algorithm.

Definition 8 (Side effects)**.** Flaws that are caused by the application of a resolver are called *related flaws*. They are inserted into the *agenda*[3] with each application of a resolver:

- *Related subgoals* are all the new open conditions inserted by new steps.
- *Related threats* are the causal links threatened by the insertion of a new step or the deletion of a guarding link.

[3] An agenda is a flaw container used for the flaw selection of POCL.

Flaws can also become irrelevant when a resolver is applied. It is always the case for the targeted flaw, but this can also affect other flaws. Those *invalidated flaws* are removed from the agenda upon detection:

- *Invalidated subgoals* are subgoals satisfied by the new causal links or the removal of their needer.
- *Invalidated threats* happen when the breaker no longer threatens the causal link because the order guards the threatened causal link or either of them has been removed.

Example: Adding the action $pour(water, cup)$ causes a related subgoal for each of the preconditions of the action which are: the cup and the water must be taken and water must not already be in the cup.

In algorithm 1 we present a generic version of POCL inspired by Ghallab *et al.* (2004, sec. 5.4.2).

Algorithm 1 Partial Order Planner

```
1  function POCL(Agenda a, Problem P)
2      if a = ∅ then                          ▷ Populated agenda needs to be provided
3          return Success                     ▷ Stops all recursion
4      Flaw f ← choose(a)                     ▷ Heuristically chosen flaw
5      Resolvers R ← solve(f, P)
6      for all r ∈ R do ▷ Non-deterministic choice operator
7          apply(r, π)                        ▷ Apply resolver to partial plan
8          Agenda a' ← update(a)
9          if POCL(a', P) = Success then      ▷ Refining recursively
10             return Success
11         revert(r, π) ▷ Failure, undo resolver application
12     a ← a ∪ {f}                            ▷ Flaw was not resolved
13     return Failure                         ▷ Revert to last non-deterministic choice
```

For our version of POCL we follow a refinement procedure that works in several generic steps. In figure 1 we detail the resolution of a subgoal as done in the algorithm 1.

The first is the search for resolvers. It is often done in two separate steps: first, select the candidates and then check each of them for validity. This is done using the polymorphic function `solve` at line 5.

In the case of subgoals, variable unification is performed to ensure the compatibility of the resolvers. Since this step is time-consuming, the operator is instantiated accordingly at this step to factories the computational effort. Composite operators have also all their methods instantiated at this step if they are selected as a candidate.

Then a resolver is picked non-deterministically for applications (this can be heuristically driven). At line 7 the resolver is effectively applied to the current plan. All side effects and invalidations are handled during the update of the agenda at line 8. If a problem occurs, line 11 backtracks and tries other resolvers. If no resolver fits the flaw, the algorithm backtracks to previous resolver choices to explore all the possible plans and ensure completeness.

Figure 1: Example of the refinement process for subgoal resolution

In definition 8, we mentioned effects that aren't present in classical POCL, namely *negative resolvers*. All classical resolvers only add steps and causal links to the partial plan. Our method needs to remove composite steps and their adjacent links when expanding them.

4 The Heart of the Method

In this section, we explain how our method combines POCL with HTN planning and how they are used to generate intermediary abstract plans.

4.1 Additional Notions

In order to properly introduce the changes made for using HTN domains in POCL, we need to define a few notions.

Transposition is needed to define decomposition.

Definition 9 (Transposition)**.** In order to transpose the causal links of an action a' with the ones of an existing step a in a plan π, we use the following operation :

$$a \rhd^-_\pi a' = \left\{ \phi^-(l) \xrightarrow{causes(l)} a' \,:\, l \in L^-_\pi(a) \right\}$$

It is the same with $a' \xrightarrow{causes(l)} \phi^+(l)$ and L^+ for $a \rhd^+ a'$. This supposes that the respective preconditions and effects of a and a' are equivalent. When not signed, the transposition is generalized: $a \rhd a' = a \rhd^- a' \cup a \rhd^+ a'$.

Example: $a \rhd^- a'$ gives all incoming links of a with the target set to a' instead.

Definition 10 (Proper Actions)**.** Proper actions are actions that are "contained" within an entity. We note this notion $A_a = A_a^{lv(a)}$ for an action a. It can be applied to various concepts :

- For a *domain* or a *problem*, $A_\mathcal{P} = A_\mathcal{D}$.
- For a *plan*, it is $A^0_\pi = S_\pi$.
- For an *action*, it is $A^0_a = \bigcup_{m \in methods(a)} S_m$. Recursively: $A^n_a = \bigcup_{b \in A^0_a} A^{n-1}_b$. For atomic actions, $A_a = \varnothing$.

Example: The proper actions of $make(drink)$ are the actions contained within its methods. The set of extended proper actions adds all proper actions of its single proper composite action $infuse(drink, water, cup)$.

Definition 11 (Abstraction Level)**.** This is a measure of the maximum amount of abstraction an entity can express:[4]

$$lv(x) = \left(\max_{a \in A_x} (lv(a)) + 1 \right) [A_x \neq \varnothing]$$

Example: The abstraction level of any atomic action is 0 while it is 2 for the composite action $make(drink)$. The example domain (in listing 1) has an abstraction level of 3.

4.2 Abstraction In POCL

The most straightforward way to handle abstraction in regular planners is illustrated by Duet (Gerevini et al., 2008) by managing hierarchical actions separately from a task insertion planner. We chose to add abstraction in POCL in a manner inspired by the work of Bechon *et al.* (2014) on a planner called HiPOP. The difference between the original HiPOP and our implementation of it is that we focus on the expressivity and the ways flaw selection can be exploited for partial resolution. Our version is lifted at runtime while the original is grounded for optimizations. All mechanisms we have implemented use POCL but with different management of flaws and resolvers. The original algorithm 1 is left untouched.

One of those changes is that resolver selection needs to be altered for subgoals. Indeed, as stated by the authors of HiPOP: the planner must ensure the selection of high-level operators in order to benefit from the hierarchical aspect of the domain. Otherwise, adding operators only increases the branching factor. Composite actions are not usually meant to stay in a finished plan and must be decomposed into atomic steps from one of their methods.

Definition 12 (Decomposition Flaws)**.** They occur when a partial plan contains a non-atomic step. This step is the needer a_n of the flaw. We note its decomposition $a_n\oplus$.

- *Resolvers*: A decomposition flaw is solved with a **decomposition resolver**. The resolver will replace the needer with one of its instantiated methods $m \in methods(a_n)$ in the plan π. This is done by using transposition such that: $a_n \oplus^m_\pi = \langle S_m \cup (S_\pi \setminus \{a\}), a_n \rhd^- I_m \cup a_n \rhd^+ G_m \cup (L_\pi \setminus L_\pi(a_n)) \rangle$.
- *Side effects*: A decomposition flaw can be created by the insertion of a composite action in the plan by any resolver and invalidated by its removal :

$$\bigcup_{a_m \in S_m}^{f \in pre(a_m)} \pi' \ddagger_f a_m \bigcup_{a_b \in S_{\pi'}}^{l \in L_{\pi'}} a_b \otimes l \bigcup_{a_c \in S_m}^{lv(a_c) \neq 0} a_c \oplus$$

Example: When adding the step $make(tea)$ in the plan to solve the subgoal that needs tea being made, we also introduce a decomposition flaw that will need this composite step replaced by its method using a decomposition resolver. In order to decompose a composite action into a plan, all existing links are transposed to the initial and goal step of the selected method, while the composite action and its links are removed from the plan. The main differences between HiPOP and HEART in our implementations are the functions of flaw selection and the handling of the results (one plan for HiPOP and a plan per cycle for HEART). In HiPOP, the flaw selection is made by prioritizing the decomposition flaws. Bechon *et al.* (2014) state that it makes the full resolution faster. However, it also loses opportunities to obtain abstract plans in the process.

4.3 Cycles

The main focus of our work is toward obtaining **abstract plans** which are plans that are completed while still containing composite actions. In order to do that the flaw selection function will enforce cycles in the planning process.

Definition 13 (Cycle)**.** A cycle is a planning phase defined as a triplet $c = \langle lv(c), agenda, \pi_{lv(c)} \rangle$ where : $lv(c)$

[4] We use Iverson brackets here, see notations in table 1.

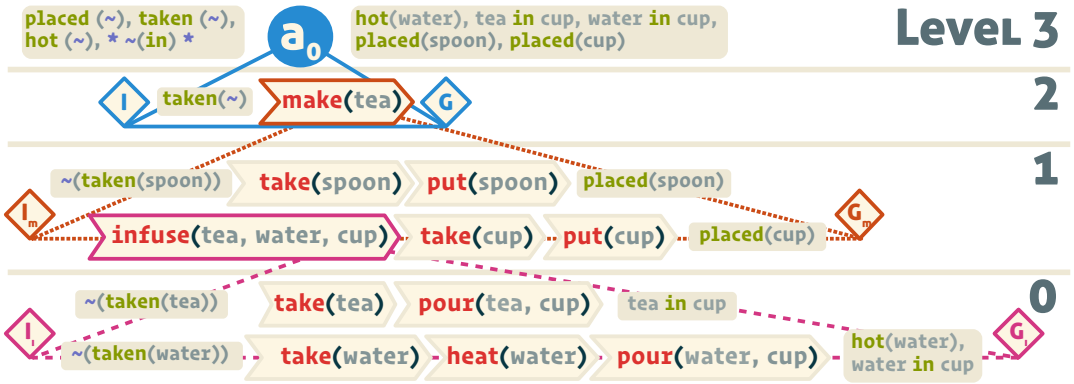

Figure 2: Illustration of how the cyclical approach is applied on the example domain. Atomic actions that are copied from a cycle to the next are omitted.

is the maximum abstraction level allowed for flaw selection in the *agenda* of remaining flaws in partial plan $\pi_{lv(c)}$. The resolvers of subgoals are therefore constrained by the following: $a_p \downarrow_f a_n : lv(a_p) \leq lv(c)$.

During a cycle all decomposition flaws are delayed. Once no more flaws other than decomposition flaws are present in the agenda, the current plan is saved and all remaining decomposition flaws are solved at once before the abstraction level is lowered for the next cycle: $lv(c') = lv(c) - 1$. Each cycle produces a more detailed abstract plan than the one before.

Abstract plans allow the planner to do an approximate form of anytime execution. At any given time the planner is able to return a fully supported plan. Before the first cycle, the plan returned is $\pi_{lv(a_0)}$.

Example: In our case using the method of intent recognition of Sohrabi *et al.* Sohrabi et al. (2016), we can already use $\pi_{lv(a_0)}$ to find a likely goal explaining an observation (a set of temporally ordered fluents). That can make an early assessment of the probability of each goal of the recognition problem.

For each cycle c, a new plan $\pi_{lv(c)}$ is created as a new method of the root operator a_0. These intermediary plans are not solutions of the problem, nor do they mean that the problem is solvable. In order to find a solution, the HEART planner needs to reach the final cycle c_0 with an abstraction level $lv(c_0) = 0$. However, these plans can be used to derive meaning from the potential solution of the current problem and give a good approximation of the final result before its completion.

Example: In the figure 2, we illustrate the way our problem instance is progressively solved. Before the first cycle c_2, all we have is the root operator and its plan π_3. Then within the first cycle, we select the composite action $make(tea)$ instantiated from the operator $make(drink)$ along with its methods. All related flaws are fixed until all that is left in the agenda is the abstract flaws. We save the partial plan π_2 for this cycle and expand $make(tea)$ into a copy of the current plan π_1 for the next cycle. The solution of the problem will be stored in π_0 once found.

5 Results

5.1 Theoretical analysis

In this section, we prove several properties of our method and resulting plans : HEART is complete, sound and its abstract plans can always be decomposed into a valid solution.

The completeness and soundness of POCL has been proven in (Penberthy et al., 1992). An interesting property of POCL algorithms is that flaw selection strategies do not impact these properties. Since the only modification of the algorithm is the extension of the classical flaws with a decomposition flaw, all we need to explore, to update the proofs, is the impact of the new resolver. By definition, the resolvers of decomposition flaws will take into account all flaws introduced by its resolution into the refined plan. It can also revert its application properly.

Lemma (Decomposing preserves acyclicity)**.** *The decomposition of a composite action with a valid method in an acyclic plan will result in an acyclic plan. Formely,* $\forall a_s \in S_\pi : a_s \not\succ_\pi a_s \implies \forall a'_s \in S_{a \oplus^m_\pi} : a'_s \not\succ_{a \oplus^m_\pi} a'_s$.

Proof. When decomposing a composite action a with a method m in an existing plan π, we add all steps S_m in the refined plan. Both π and m are guaranteed to be cycle free by definition. We can note that $\forall a_s \in S_m : (\nexists a_t \in S_m : a_s \succ a_t \wedge \neg f \in eff(a_t)) \implies f \in eff(a)$. Said otherwise, if an action a_s can participate a fluent f to

the goal step of the method m then it is necessarily present in the effects of a. Since higher level actions are preferred during the resolver selection, no actions in the methods are already used in the plan when the decomposition happens. This can be noted $\exists a \in \pi \implies S_m \cup S_\pi$ meaning that in the graph formed both partial plans m and π cannot contain the same edges therefore their acyclicity is preserved when inserting one into the other.

□

Lemma (Solved decomposition flaws cannot reoccur)**.** *The application of a decomposition resolver on a plan π, guarantees that $a \notin S_{\pi'}$ for any partial plan refined from π without reverting the application of the resolver.*

Proof. As stated in the definition of the methods (definition 4): $a \notin A_a$. This means that a cannot be introduced in the plan by its decomposition or the decomposition of its proper actions. Indeed, once a is expanded, the level of the following cycle $c_{lv(a)-1}$ prevents a to be selected by subgoal resolvers. It cannot either be contained in the methods of another action that are selected afterward because otherwise following definition 11 its level would be at least $lv(a) + 1$.

□

Lemma (Decomposing to abstraction level 0 guarantees solvability)**.** *Finding a partial plan that contains only decomposition flaws with actions of abstraction level 1, guarantees a solution to the problem.*

Proof. Any method m of a composite action $a : lv(a) = 1$ is by definition a solution of the problem $\mathcal{P}_a = \langle \mathcal{D}, C_{\mathcal{P}}, a \rangle$. By definition, $a \notin A_a$, and $a \notin A_{a \oplus_\pi^m}$ (meaning that a cannot reoccur after being decomposed). It is also given by definition that the instantiation of the action and its methods are coherent regarding variable constraints (everything is instantiated before selection by the resolvers). Since the plan π only has decomposition flaws and all flaws within m are guaranteed to be solvable, and both are guaranteed to be acyclical by the application of any decomposition $a \oplus_\pi^m$, the plan is solvable.

□

Lemma (Abstract plans guarantee solvability)**.** *Finding a partial plan π that contains only decomposition flaws, guarantees a solution to the problem.*

Proof. Recursively, if we apply the previous proof on higher level plans we note that decomposing at level 2 guarantees a solution since the method of the composite actions are guaranteed to be solvable.

□

From these proofs, we can derive the property of soundness (from the guarantee that the composite action provides its effects from any methods) and completeness (since if a composite action cannot be used, the planner defaults to using any action of the domain).

5.2 Experimental evaluation

In order to assess its capabilities, HEART was evaluated on two criteria: quality and complexity. All tests were executed on an Intel® Core™ i7-7700HQ CPU clocked at 2.80GHz. The Java process used only one core and was not limited by time or memory.[5] Each experiment was repeated between 700 and 10 000 times to ensure that variations in speed were not impacting the results.

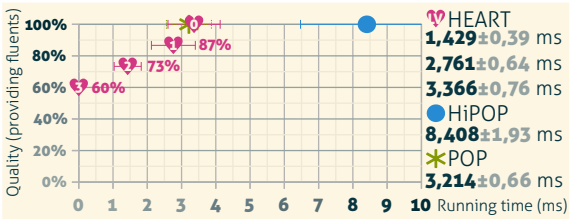

Figure 3: Evolution of the quality with computation time.

Figure 3 shows how the quality is affected by the abstraction in partial plans. The tests are made using our example domain (see listing 1). The quality is measured by counting the number of providing fluents in the plan $\left|\bigcup_{a \in S_\pi} eff(a)\right|$. This metric is actually used to approximate the probability of a goal given observations in intent recognition ($P(G|O)$ with noisy observations, see (Sohrabi et al., 2016)). The percentages are relative to the total number of unique fluents of the complete solution. These results show that in some cases it may be more interesting to plan in a leveled fashion to solve HTN problems. For the first cycle of level 3, the quality of the abstract plan is already of 60%. This is the quality of the exploitation of the plan *before any planning*. With almost three-quarters of the final quality and less than half of the complete computation time, the result of the first cycle is a good quality/time compromise.

In the second test, we used generated domains. These domains consist of an action of abstraction level 5. This action has a single method containing a number of actions of level 4. We call this number the width of the domain. All needed actions are built recursively to form a tree shape. Atomic actions only have single fluent effects. The goal is the effect of the higher level action and the

[5]The source code of HEART will be available at genn.io/-heart.

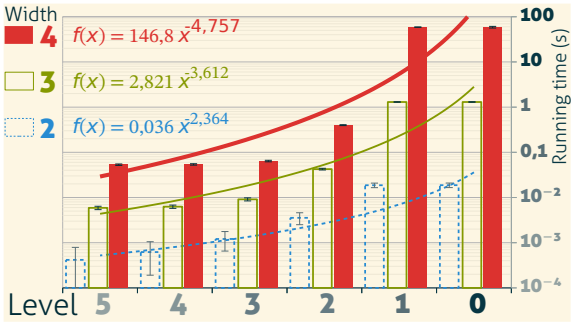

Figure 4: Impact of domain shape on the computation time by levels. The scale of the vertical axis is logarithmic. Equations are the definition of the trend curves.

initial state is empty. These domains do not contain negative effects. Figure 4 shows the computational profile of HEART for various levels and widths. We note that the behavior of HEART seems to follow an exponential law with the negative exponent of the trend curves seemingly being correlated to the actual width. This means that computing the first cycles has a complexity that is close to being *linear* while computing the last cycles is of the same complexity as classical planning which is at least *P-SPACE* (depending on the expressivity of the domain) (Erol et al., 1995).

Conclusion

In this paper, we have presented a new planner called HEART based on POCL. An updated planning framework fitting the need for such a new approach was proposed. We showed how HEART performs compared to complete planners in terms of speed and quality. While the abstract plans generated during the planning process are not complete solutions, they are exponentially faster to generate while retaining significant quality over the final plans. They are also proof of solvability of the problem. By using these plans, it is possible to generate explanations of intractable problems within tight time constraints.

References

Baioletti, M., Marcugini, S., and Milani, A. (1998). Encoding planning constraints into partial order planning domains. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 608–616. Morgan Kaufmann Publishers Inc.

Baker, C. L., Tenenbaum, J. B., and Saxe, R. R. (2007). Goal inference as inverse planning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*,Vol. 29.

Bechon, P., Barbier, M., Infantes, G., Lesire, C., and Vidal, V. (2014). HiPOP: Hierarchical Partial-Order Planning. In *European Starting AI Researcher Symposium*,Vol. 264, pages 51–60. IOS Press.

Beckett, D., and Berners-Lee, T. (2011). *Turtle - Terse RDF Triple Language* (W3C Team Submission). W3C.

Bercher, P., Keen, S., and Biundo, S. (2014). Hybrid planning heuristics based on task decomposition graphs. In *Seventh Annual Symposium on Combinatorial Search*.

Biundo, S., and Schattenberg, B. (2001). From abstract crisis to concrete relief preliminary report on flexible integration on nonlinear and hierarchical planning. In *Proceedings of the European Conference on Planning*.

Chen, J., Chen, Y., Xu, Y., Huang, R., and Chen, Z. (2013). A Planning Approach to the Recognition of Multiple Goals. *International Journal of Intelligent Systems*, *28*(3), 203–216.

Dvorak, F., Bit-Monnot, A., Ingrand, F., and Ghallab, M. (2014). A flexible ANML actor and planner in robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*.

Erol, K., Hendler, J., and Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *AAAI*,Vol. 94, pages 1123–1128.

Erol, K., Nau, D. S., and Subrahmanian, V. S. (1995). Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, *76*(1-2), 75–88.

Fox, M. (1997). Natural hierarchical planning using operator decomposition. In *European Conference on Planning*, pages 195–207. Springer.

Fox, M., Long, D., and Magazzeni, D. (2017). Explainable Planning. In *Proceedings of IJCAI Workshop on Explainable AI*. Melbourne, Australia.

Gerevini, A., Kuter, U., Nau, D. S., Saetti, A., and Waisbrot, N. (2008). Combining Domain-Independent Planning and HTN Planning: The Duet Planner. In *Proceedings of the European Conference on Artificial Intelligence*,Vol. 18, pages 573–577.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated planning: Theory & practice*. Elsevier.

Ghallab, M., Nau, D., and Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

Göbelbecker, M., Keller, T., Eyerich, P., Brenner, M., and Nebel, B. (2010). Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proceedings of the International Conference on Automated Planning and Scheduling*,Vol. 20, pages 81–88. AAAI Press.

Kambhampati, S., Mali, A., and Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *AAAI/IAAI*, pages 882–888.

Lemai, S., and Ingrand, F. (2004). Interleaving temporal planning and execution in robotics domains. In *AAAI*,Vol. 4, pages 617–622.

Marthi, B., Russell, S. J., and Wolfe, J. A. (2007). Angelic Semantics for High-Level Actions. In *ICAPS*, pages 232–239.

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *J. Artif. Intell. Res.(JAIR)*, *20*, 379–404.

Penberthy, J. S., Weld, D. S., and others. (1992). UCPOP: A Sound, Complete, Partial Order Planner for ADL. *Kr*, *92*, 103–114.

Ramirez, M., and Geffner, H. (2009). Plan recognition as planning. In *Proceedings of the International Conference on International Conference on Automated Planning and Scheduling*,Vol. 19, pages 1778–1783. AAAI Press.

Ramirez, M., and Geffner, H. (2010). Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the Conference of the Association for the Advancement of Artificial Intelligence*,Vol. 24, pages 1121–1126.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*(2), 115–135.

Sacerdoti, E. D. (1975). *The nonlinear nature of plans*. STANFORD RESEARCH INST MENLO PARK CA.

Seegebarth, B., Müller, F., Schattenberg, B., and Biundo, S. (2012). Making hybrid plans more clear to human usersa formal approach for generating sound explanations. In *Proceedings of the Twenty-Second International Conference on International Conference on Automated Planning and Scheduling*, pages 225–233. AAAI Press.

Sohrabi, S., Riabov, A. V., and Udrea, O. (2016). Plan Recognition as Planning Revisited. In *Proceedings of the International Joint Conference on Artificial Intelligence*,Vol. 25.

Sreedharan, S., Chakraborti, T., and Kambhampati, S. (2018). Handling Model Uncertainty and Multiplicity in Explanations via Model Reconciliation. In *International Conference on Automated Planning and Scheduling*.

Sreedharan, S., Srivastava, S., and Kambhampati, S. (2018). Hierarchical Expertise-Level Modeling for User Specific Robot-Behavior Explanations. In *International Joint Conference on Artificial Intelligence*.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, *15*(4), 27.

Young, R. M., and Moore, J. D. (1994). DPOCL: A principled approach to discourse planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 13–20. Association for Computational Linguistics.