Advanced User Assistance Based on AI Planning

Susanne Biundo, Pascal Bercher, Thomas Geier, Felix Müller, Bernd Schattenberg

⟨firstname⟩.⟨lastname⟩@uni-ulm.de

Institute of Artificial Intelligence, Ulm University, 89069 Ulm, Germany

Abstract

Artificial Intelligence technologies enable the implementation of cognitive systems with advanced planning and reasoning capabilities. This article presents an approach to use hybrid planning – a method that combines reasoning about procedural knowledge and causalities – to provide user-centered assistance.

Based on a completely declarative description of actions, tasks, and solution methods, hybrid planning allows for the generation of knowledge-rich plans of action. The information those plans comprise includes causal dependencies between actions on both abstract and primitive levels as well as information about their hierarchical and temporal relationships.

We present the hybrid planning approach in detail and show its potential by describing the realization of various assistance functionalities based on complex cognitive processes like the generation, repair, and explanation of plans. Advanced user assistance is demonstrated by means of a practical application scenario where an innovative electronic support mechanism helps a user to operate a complex mobile communication device.

Keywords: cognitive technical systems, companion-technology, hybrid planning, plan repair, plan explanation, real-world planning

1. Introduction

In professional and private daily business and especially when pursuing any major undertaking, strategic planning, reasoning about the consequences of acting, and weighing the pros and cons of various options are crucial cognitive capabilities human beings routinely exhibit. When aiming at the construction of advanced intelligent systems that assist users in high-level tasks and support their decision making, it seems therefore quite natural and adequate to rely on a technical equivalent of those cognitive capabilities.

The field of Artificial Intelligence (AI) Planning provides a large variety of methods to plan and reason and to do so by taking specific characteristics of prospective applications and environments into account [1]. Its most popular realm nowadays is classical state-based planning that originates from early work by Fikes and Nilsson [2]. The publication of the *Graphplan* algorithm in the mid-1990s [3] and the setup of the *International Planning Competition* [4] revived the area and launched a strong development towards heuristic forward-search-based planning [5, 6, 7]. While state-based approaches aim at the generation of linear action sequences that are intended to be automatically executed by systems, there are two main strands in the field dedicated to the construction of more elaborate plan structures and to explicitly reflecting the kinds of reasoning humans perform when developing plans. In partial-order causal-link (POCL) planning [8, 9], plans are partially ordered sets of actions and show causal dependencies between actions explicitly. This allows for flexibility w.r.t. the order in which actions are finally executed and enables a human user to grasp the causal structure of the plan and to understand why certain actions are part of it. Hierarchical task network (HTN) planning [10, 11] features another important principle of intelligent planning, namely abstraction. It allows for the specification of both complex abstract tasks and predefined standard solutions for these tasks. Here, plan generation is a top-down refinement process that stepwise replaces abstract tasks by appropriate (abstract) solution plans until an executable action sequence is obtained. HTN planning is particularly useful for solving real-world planning problems since it provides the means to immediately reflect and employ the abstraction hierarchies that are inherent in many domains.

By combining the characteristic features of POCL and HTN techniques *hybrid planning* [12, 13, 14, 15] smoothly integrates reasoning about procedural knowledge and causalities, thereby providing an enabling technology for the realization of complex cognitive capabilities of technical systems. Based on a completely declarative description of actions, tasks, and solution methods, hybrid planning allows for the generation of knowledge-rich plans of action. The information those plans comprise includes causal dependencies between actions on both abstract and primitive levels as well as information about their hierarchical and temporal relationships. By making use of this information, as well as of the underlying declarative domain models, complex cognitive capabilities like the generation of courses of action on various abstraction levels, the stable repair of failed plans in response to some unexpected environment changes, and the explanation of different solutions for a given planning problem – to name just a few – can be implemented by advanced automated reasoning techniques.

In this article, we present the potential of hybrid planning in view of complex cognition by describing the realization of various assistance functionalities for individual users of a technical system. They include (1) generating a plan for a specific user and advising him to carry out the plan in order to achieve a current task, (2) instructing the user on how to escape from a situation where the execution of this plan unexpectedly failed, and (3) justifying and explaining the proposed solution plans in an adequate manner. These assistance functionalities can be provided by a component that relies on a domain-independent hybrid planner, a plan repair component, and a plan explanation facility.

In the scenario we have chosen to illustrate our approach, such an assistance component appears as an innovative electronic support mechanism that

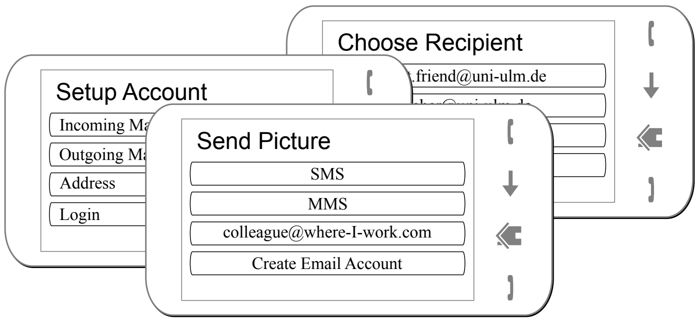

Figure 1: A schematic view of a commercially available smart phone.

smoothly helps a human user to operate his new mobile phone. In applications where the domain is a technical device, like in this scenario, setting up the planning domain model is rather straightforward. It includes the representation of relevant states and of actions that cause state transitions. Actions are applicable if certain prerequisites hold; their effects specify properties of the successor state.

The state of the phone includes aspects like the currently displayed menu and the pictures saved on the phone, but also whether it currently has reception. Our model does not account for every detail of the phone, however. Aspects below a certain level of detail are not relevant for providing support, because they are not relevant for operating the phone. For example, we do not model how the cell phone internally handles the communication with the network or signal strength; we only distinguish between having reception and not having reception.

The state of the phone can be changed by performing actions like pressing a button, activating the camera, and the like. Figure 1 shows a schematic view of the mobile phone. Performing an action like pressing the touch-screen button labeled with "MMS" changes the state of the phone. Before the button is pressed, the state of the cell phone is that it displays the send menu for pictures; afterwards, it displays the MMS dialog menu where the user can enter the details of the message to send.

As we are dealing with a deterministic technical system, all actions have a well-defined outcome. This allows us to predict future states of the mobile phone: if we know the state of the phone and the action we want to apply, we can determine its state after the application of the action. In addition, we again abstract from unnecessary details and treat action application as atomic. We thus do not need an explicit representation of time and assume that actions change the state of the phone instantaneously.

Performing a single action is usually not enough to achieve a given objective; a combination of various actions is required instead. The order in which these actions are performed generally matters. For instance, one cannot send an MMS before having entered a recipient. In some cases however, the order may not be relevant: it does not matter in which order one fills out the fields of a form, for example. Hence, the required actions are partially ordered. The particular order in which the actions have to be executed is determined by their enabling preconditions and their effects and with that imposed by causal dependencies between the actions. As a consequence, it seems to be obvious that POCL plans are an adequate means to represent the operation of a technical device.

Moreover, there are higher-level operations or tasks that comprise a whole bunch of simpler actions, an example being sending an MMS. It consists of selecting a recipient, typing in the message, and so forth. In order to perform this task, the user needs to navigate to the MMS menu, enter the recipient and various types of content, and finally has to press the send button. Those higher-level operations are not limited to a single realization; in general, there are various ways to perform them. Thus, it seems natural to use HTN representation means when dealing with these kinds of operations.

The article continues with the introduction of our formal planning framework. Section 3 presents the hybrid planning algorithm and its formal properties. Our formalizations and the respective algorithms are illustrated by a running example where a person is supported in using his new and yet unfamiliar mobile phone. In Section 4 we address the problem of unexpected environment changes and show how our plan repair mechanism is applied to help the user escaping from a situation where his intended action is doomed to failure. After that, we discuss the challenge to come up with adequate explanations of plan-based user instructions. Section 5 describes the ways in which both the causal structure of plans and the hybrid plan generation process from which they originate can be employed to achieve this objective. Finally, our presentation ends with some concluding remarks in Section 6.

2. The Hybrid Planning Framework

Hybrid planning – the fusion of partial-order causal-link (POCL) planning and hierarchical task network (HTN) planning – combines the advantages of two different approaches for solving planning problems.

POCL planning is a technique used for solving state-based planning problems. The objective is to accomplish some desired property of the world, i. e., to reach some goal state by applying actions in a correct order starting in a given initial state.

As mentioned, POCL planning is quite suitable for the purpose of finding plans that are intended to be executed by human users, since, on the one hand, the explicit information about causality helps to understand the plan and, on the other hand, the partial order of actions allows the user for more degrees of freedom in selecting the next action to execute.

HTN planning extends the principle of state-based planning to a hierarchy on the available actions. In HTN planning, actions are generally referred to as tasks. This hierarchy is established using so-called primitive tasks with preconditions and effects like in pure state-based planning, as well as abstract tasks that do not have any precondition or effect, but solely serve as containers for plans that represent predefined implementations. However, these implementations can again contain abstract tasks. The mapping between an abstract task and the plan implementing it is done by so-called decomposition methods. Thus, for each abstract task there is at least one method defining its implementation. A crucial difference between state-based and HTN planning is the solution criterion. Whereas the goal in state-based planning is to achieve a desired property, no matter which actions have to be used to accomplish this, the goal in HTN planning is to find a plan that is a valid decomposition of the initial abstract task, such that the resulting plan only contains primitive tasks.

The top-down manner in which HTN planning systems search for plans is similar to planning performed by humans when planning to achieve complex tasks like planning a business trip to a conference or setting up a complex technical device.

Most real-world application domains, like emergency evacuation and crisis management [1, Chapter 22][14] and transportation/logistics problems [16], make use of hierarchical structures on tasks and resources to a very high extent. HTN planning techniques are therefore essential to tackle those problems. Traditional HTN planning can only come up with solutions that are modeled in advance by means of decompositions for abstract tasks. In fact, HTN planning is intended to allow only these solutions, since they have been carefully designed by domain experts. We regard the procedural knowledge given in terms of decomposition methods as an enrichment of the domain model (i. e., the encoding of the world), rather than as a restriction to a subset of valid plans. Therefore, we pursue the integration of HTN planning and state-based planning, called *hybrid planning* [12, 13, 14, 15], which turned out to be well suited for solving real-world problems [17, 18].

2.1. Logical Language

Our hybrid planning formalism relies on an order-sorted, quantifier-free predicate logic. Due to space limitations we omit the definition of its *semantics* and refer to our previous work for any further details [14]. Its *syntax* is based on the *logical language* $\mathtt{L} = \langle Z, <, R, C, V, L \rangle$.

In sorted logics, all variables and constants are of some sort $z \in Z$. Table 1 lists the sorts used in our example domain model. Order-sorted logics additionally impose a hierarchy on sorts which allows for more adequate and concise formalizations. The relation $<$ is used to express this hierarchy on the sort symbols in Z. Figure 2 shows a graphical representation of a part of the sort hierarchy of our domain model. R is a Z^*-indexed family of finite disjoint sets of *relation symbols*, which are used to express properties of objects in the real world. Accordingly, C is a Z-indexed family of finite disjoint sets of *constant symbols* that represent objects in the real world. If we want to model that a specific email message is associated with a specific recipient, we use a constant like `EMAIL42` of sort Email to represent this email and a constant like `CONTACT23` of sort Contact to represent its recipient. The expression `RecipientIsSet(EMAIL42, CONTACT23)` then states the desired association. The

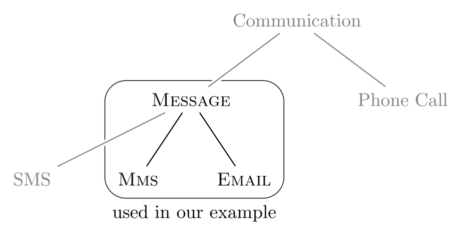

Figure 2: Part of the sort hierarchy of our smart phone domain model.

expression `Mode`(`USINGCAMERA`) encodes that the cell phone is in a mode in which pictures can be taken. V is a Z-indexed family of infinite disjoint sets of *variable symbols*; in our examples, variable symbols are written with a preceding question mark to distinguish them from constant symbols. To refer to the set of variables and constants of a sort $z \in Z$, we write V_z and C_z, respectively. Finally, L is an infinite set of labels used for identifying different occurrences of identical tasks.

2.2. Tasks and Plans

A *task* t is a tuple $\langle pre, post\rangle$, specifying a *precondition* and a *postcondition*. Pre- and postconditions are sets of literals over the relations of the logical language $\mathtt{L}$ and depend on the task parameters $\bar{\tau}(t) = \tau_1(t) \ldots \tau_{n(t)}(t)$, where $n(t)$ is the length of this sequence, also called the arity of task t. For convenience, we also write $t(\bar{\tau})$ to refer to a task t. For example, the task `SetUpEmail-Account`(`?acc1`) $= \langle\{$`Mode`(`SENDMENU`), $\neg$`SetUp`(`?acc1`)$\}, \{$`SetUp`(`?acc1`)$\}\rangle$ states that setting up an email account requires the cell phone to be in the appropriate mode and that the account is not set up already; as a result, the respective email account is set up and can hence be used. We collect in $post^+(t)$ and $post^-(t)$ the atoms that occur positively and negatively in the postcondition of task t and denote them as the *positive* and *negative effects* of t, respectively.

A *state* is a finite set of ground atoms. A task t with ground pre- and postconditions is called *applicable* in a state s, if the positive literals of its precondition are contained in s and the negative ones are not. If t is applicable in a state s, its application leads to the state $s' = (s \setminus post^-(t)) \cup post^+(t)$ and is undefined, otherwise. The applicability of sequences of ground tasks is defined inductively over state sequences as usual.

Let the state of the mobile phone be $s = \{$`Mode`(`SENDMENU`), `SetUp`(`ACC1`)$\}$. The task `SetUpEmailAccount` with its parameter `?acc1` being associated with the constant `ACC2` is applicable in s, since all positive literals of its precondition $\{$`Mode`(`SENDMENU`), $\neg$`SetUp`(`?acc1`)$\}$ are contained in s and the negative ones are not. Its application leads to the state $s' = s \cup \{$`SetUp`(`ACC2`)$\}$.

A *plan* or *task network* is a tuple $P = \langle TE, \prec, VC, CL\rangle$ consisting of a finite set TE of *plan steps* or *task expressions* $te = l : t$, where t is a (partially) grounded task and $l \in L$ is a unique label to distinguish different occurrences of the same task within the same plan.

Table 1: A subset of elements of the logical language `L`, which we use to model the functionality of a smart phone.

Sorts Z

Name	Description
MODE	constants of this sort are used to represent the possible modes of the cell phone
MESSAGE	messages
MMS	MMS messages
EMAIL	email messages
PICTURE	pictures
CONTACT	address book entries
ACCOUNT	configurable email accounts

Relations R

Name	Signature	Description
`RecipientIsSet`	MESSAGE × CONTACT	associates a message, i. e., an MMS or email with a recipient
`Stored`	PICTURE	true iff a picture is stored
`Mode`	MODE	the active mode
`HasReception`		true iff the phone has reception
⋮		

Constants C

Name	Signature	Description
`USINGCAMERA`	MODE	the mode in which pictures can be taken
`SHOWALBUM`	MODE	the mode for displaying the stored pictures
`ACC1`,`ACC2`,...	ACCOUNT	email accounts
`PIC1`,`PIC2`,...	PICTURE	pictures
⋮		

CL is a set of *causal links*, each of which has the form $\langle l_i, \phi, l_j \rangle$, indicating that the task expression te_i provides the task expression te_j with its precondition $\phi \in post(te_i) \cap pre(te_j)$, where $post(te)$ and $pre(te)$ refer to the post- and precondition of t, if $te = l : t$. This explicit representation of causal relationships between tasks is, next to our integration of hierarchical concepts, one of our main arguments for the adequacy of our formalism in our problem setting of providing assistance to human users in scenarios which involve planning capabilities because they are direct justifications for the occurrence of tasks.

Every causal link $\langle l_i, \phi, l_j \rangle$ implicitly induces an ordering between the plan steps with labels l_i and l_j. Additionally, the set of ordering constraints $\prec$ contains explicit orderings that are predefined by the model or are added as a result of the planning process. We write $\prec^*_{\mathrm{CL}}$ in infix notation when we refer to the transitive closure imposed both by the constraints from $\prec$ and those induced by *CL*. Note that $\prec^*_{\mathrm{CL}}$ defines the partial order that governs the execution of the plan – the ordering constraints $\prec$ alone are not enough.

The set of *variable constraints VC* is a set of *(non-)co-designations* used for grounding tasks and to force (in-)equality between variables. Formally, for two tasks t and t', $\tau_i(t) \doteq \tau_j(t')$ constrains $\tau_i(t)$ and $\tau_j(t')$ to be identical and, for co-designating variables with constants, $\tau_i(t) \doteq c$ constrains the variable $\tau_i(t)$ to be equal to the constant $c \in C_z$, where $z \in Z$ is the sort of c. *Non-co-designations* are defined analogously.

A *causal threat* is the situation in which the partial order of a plan would allow the plan step te_k with the postcondition $\neg\psi$ to be ordered between two plan steps te_i and te_j for which there is a causal link $\langle l_i, \phi, l_j \rangle$ such that under the current variable constraints, ϕ and ψ can be unified. This situation is a threat to the causal link, because the postcondition $\neg\phi$ would falsify the formula ϕ, thus destroying the causal link. Causal threats can be resolved by promotion or demotion (i. e., by inserting an ordering constraint $te_j \prec te_k$ or $te_k \prec te_i$, respectively), by separation (i. e., by inserting a variable constraint such that ϕ and ψ cannot unify anymore), and by expansion (because the causal threat might only exist on an abstract level; cf. Section 2.3).

Example for a Plan

Figure 3 shows a plan P_{setup} for setting up an email account in the mobile phone. It uses the following tasks:

$$\begin{aligned}
\texttt{PressEmailSetup}() &= \langle \{\texttt{Mode(SENDMENU)}\}, \\
&\quad\ \{\neg\texttt{Mode(SENDMENU)}, \texttt{Mode(EMAILSETUP)}\}\rangle \\
\texttt{InputServerInfo(?acc2)} &= \langle \{\texttt{Mode(EMAILSETUP)}, \neg\texttt{SetUp(?acc2)}\}, \\
&\quad\ \{\texttt{InfoEntered(?acc2)}\}\rangle \\
\texttt{InputCredentials(?acc3)} &= \langle \{\texttt{Mode(EMAILSETUP)}, \neg\texttt{SetUp(?acc3)}\}, \\
&\quad\ \{\texttt{CredentialsEntered(?acc3)}\}\rangle
\end{aligned}$$

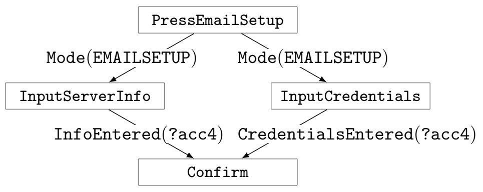

Figure 3: The plan P_{setup}, which describes how an email account can be set up in the mobile phone.

$$\begin{aligned}\texttt{Confirm(?acc4)} = \langle\{&\texttt{Mode(EMAILSETUP)}, \texttt{InfoEntered(?acc4)},\\ &\texttt{CredentialsEntered(?acc4)},\\ &\neg\texttt{SetUp(?acc4)}\},\\ \{&\neg\texttt{Mode(EMAILSETUP)}, \texttt{Mode(SENDMENU)},\\ &\texttt{SetUp(?acc4)}\}\rangle\end{aligned}$$

$P_{\text{setup}} = \langle TE_{\text{setup}}, \prec_{\text{setup}}, VC_{\text{setup}}, CL_{\text{setup}}\rangle$ is a plan where:

$$\begin{aligned} TE_{\text{setup}} &= \{l_1 : \texttt{PressEmailSetup()}, l_2 : \texttt{InputServerInfo(?acc2)},\\ &\qquad l_3 : \texttt{InputCredentials(?acc3)}, l_4 : \texttt{Confirm(?acc4)}\}\\ \prec_{\text{setup}} &= \emptyset\\ VC_{\text{setup}} &= \{\texttt{?acc2} \doteq \texttt{?acc3}, \texttt{?acc2} \doteq \texttt{?acc4}\}\\ CL_{\text{setup}} &= \{\langle l_1, \texttt{Mode(EMAILSETUP)}, l_2\rangle,\\ &\qquad \langle l_1, \texttt{Mode(EMAILSETUP)}, l_3\rangle,\\ &\qquad \langle l_2, \texttt{InfoEntered(?acc4)}, l_4\rangle,\\ &\qquad \langle l_3, \texttt{CredentialsEntered(?acc4)}, l_4\rangle\}\end{aligned}$$

Please note that the set $\prec$ of explicit ordering constraints is empty, the causal links however impose a partial order depicted in Figure 3. We include explicit ordering constraints between tasks if causal threats need to be resolved (by promotion or demotion) or if they are already present in a predefined plan of the domain model.

This plan describes which actions need to be taken in order to set up an email account. The first task is `PressEmailSetup`, which corresponds to the action of pressing the menu entry *Email Setup*. For this being possible, the mobile phone has to be in the state that actually shows the necessary menu, which is encoded by the precondition `Mode(SENDMENU)` of the task. After the execution of this action, the user has to enter the server information and the credentials, which can be done in an arbitrary order. Finally, the setup will be completed by pressing the confirmation button.

2.3. Domain Model

A *domain model* for hybrid planning is a tuple $D = \langle \mathtt{L}, \mathtt{T}, \mathtt{M} \rangle$, consisting of the logical language `L`, a set `T` of tasks and a set `M` of decomposition methods. We call a task *abstract* if there is at least one method specifying a decomposition for this task, otherwise we call it *primitive*. As an example for an abstract task, recall the task `SetUpEmailAccount(?acc1)` introduced in Section 2.2. Abstract tasks have pre- and postconditions like primitive tasks, but they do not correspond to single actions in the real world and are hence not *directly* executable by human users (they may be understandable, though; this issue will be addressed in Section 5). Instead, abstract tasks serve as containers for plans that require and achieve the pre- and postconditions of this abstract task and can thus be regarded as predefined standard solutions.

A *method* $m = \langle t, VC, P \rangle \in \mathtt{M}$ maps an abstract task t to a plan P that implements t. Each method contains a set of additional variable constraints VC to relate variables in t with variables occurring in the task network P. A method m is applied to a plan P' by replacing the abstract task t in P' by its implementation P and by inserting the variable constraints VC into the variable constraints of the plan P'. Concerning the abstract task `SetUpEmail-Account`, there is only one method that specifies how to decompose it: $m = \langle \mathtt{SetUpEmailAccount(?acc1)}, \{\mathtt{?acc1} \doteq \mathtt{?acc2}\}, P_{\text{setup}} \rangle$. This method specifies that the task `SetUpEmailAccount` can be implemented by the task network P_{setup}, that we have already seen (cf. Figure 3). The variable co-designation $\mathtt{?acc1} \doteq \mathtt{?acc2}$ ensures that the variable `?acc1` of the abstract task is properly identified with the corresponding variables used by P_{setup}. Figure 4 gives a listing of all tasks and their corresponding decomposition structure of the smart phone domain.

2.4. Problems and Solutions

A *hybrid planning problem* $\pi = \langle D, P_{\text{init}} \rangle$ consists of a *domain model* D and an *initial plan* P_{init}. The initial plan does additionally contain two artificial task expressions te_{init} and te_{goal} which are used to encode an initial and goal state, respectively. The task te_{init} has the initial state as effect and the task te_{goal} has the desired goal state as precondition. All other tasks are always ordered in between.

A plan $P_{\text{sol}} = \langle TE_{\text{sol}}, \prec_{\text{sol}}, VC_{\text{sol}}, CL_{\text{sol}} \rangle$ is a *solution* to a planning problem $\pi = \langle D, P_{\text{init}} \rangle$ if the following criteria hold:

1. P_{sol} is a refinement of P_{init}. Informally, we call a plan a refinement of P_{init}, if it results from applying modifications to it. A modification is the insertion of a plan element, i. e., an element from the set of task expressions, temporal orderings, variable constraints and causal links. The only modification that is not a pure insertion is the application of a method: it replaces an abstract task by an implementing task network and adapts the variable constraints and causal links. The formal description of the modification is introduced in the next section.
2. P_{sol} contains only primitive tasks.

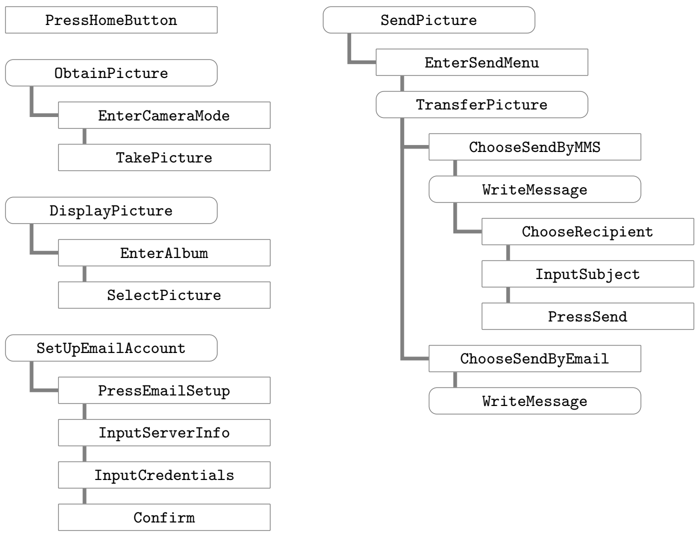

Figure 4: This figure lists the tasks inside our example domain model. It depicts the decomposition methods of abstract tasks by means of an angled line, while the subtasks of one method are connected together by straight lines.

3. All preconditions of all plan steps in P_{sol} are supported by a causal link, i. e., for each precondition ϕ of a plan step $te_j \in TE_{\text{sol}}$ there exists a causal link $\langle l_i, \phi, l_j \rangle \in CL_{\text{sol}}$ with $te_i \in TE_{\text{sol}}$.
4. There are no causal threats.
5. The ordering and variable constraints in P_{sol} are consistent, i. e., there is no plan step $te \in TE_{\text{sol}}$, such that $te \prec^*_{\text{CL}} te$ and no $v \in V_z$ for $z \in Z$, such that $VC \models v \dot{\neq} v$.
6. All tasks in P_{sol} are grounded. That is, all variables are co-designated to some constant.

Solution criterion (1) is inherited from HTN planning. In this approach, any solution must be a decomposition of the initial plan P_{init}. This HTN solution criterion is reflected in hybrid planning by the requirement that solutions must be refinements of P_{init}. Since abstract tasks are regarded as non-executable, criterion (2) ensures that only executable tasks, i. e., primitive tasks, are contained in solution plans. Criterion (3) ensures the applicability of tasks in a plan: in order for a task to be applicable in a state s, all its literals of its precondition must hold in s, what can be ensured by establishing appropriate causal links. (4) guarantees that all plan steps in all linearizations of a plan are applicable in the sense of criterion (3): causal threats can cause a literal of the precondition of a plan step to be false in some linearizations although it is supported by a causal link. Since we require *all* linearizations of a plan to be valid solutions, causal threats have to be eliminated. (5) is obviously necessary for constituting meaningful plans since neither can a task be ordered before itself nor can a

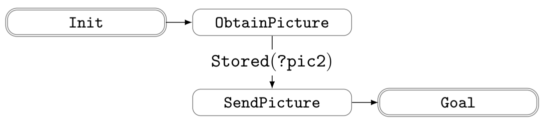

Figure 5: The initial plan P_{init} of the planning problem of our running example. It encodes James' request to take a picture and send it to a contact in his address book by the means of two abstract tasks. The initial plan contains a causal link between those tasks, because a picture has to be stored on the device, before it can be sent.

constant be different from itself. (6) maps the variables used by tasks onto the objects available in the modeled world.

Finally, we start our running example of providing plan-based support to a user. To this end, we introduce James K., the protagonist of our story.

James K. is participating in a conference on cognitive systems research. He has just attended a very interesting talk about modeling trials using ACT-R. During the session, the speaker took notes on the blackboard and James would like to discuss those with his colleagues at home. Fortunately, his new mobile phone has an integrated digital camera that seems to be up to the task. Unfortunately, the next talk is starting soon and James isn't very familiar with all those fancy functionalities, yet. Luckily, the phone is provided with an automated assistance component, on which he can rely.

This component has knowledge of the internal state of the phone and its available functions. Additionally, James can use it to query for help on sending a picture of the blackboard to one of his colleagues.

The assistance component, based on hybrid planning technology, has thus created the planning problem $\pi = \langle D, P_{\text{init}} \rangle$, with $D = \langle \mathtt{L}, \mathtt{T}, \mathtt{M} \rangle$ being the domain model and P_{init} being the initial plan, formalized as follows and illustrated in Figure 5. The initial plan P_{init} of the planning problem consists of the following tasks:

$$\begin{aligned}
\mathtt{Init}() &= \langle \emptyset, \{\mathtt{Mode}(\mathtt{INIT}), \mathtt{HasMobileNumber}(\mathtt{CONTACT1}), \\
&\qquad\quad \mathtt{HasEmail}(\mathtt{CONTACT1}), \mathtt{HasReception}(), \\
&\qquad\quad \mathtt{HasWlanConnection}()\}\rangle \\
\mathtt{ObtainPicture}(\mathtt{?pic1}) &= \langle \emptyset, \{\mathtt{Stored}(\mathtt{?pic1})\}\rangle \\
\mathtt{SendPicture}(\mathtt{?pic2}) &= \langle \{\mathtt{Stored}(\mathtt{?pic2})\}, \emptyset\rangle \\
\mathtt{Goal}() &= \langle \emptyset, \emptyset \rangle
\end{aligned}$$

The tasks `Init` and `Goal` are artificial tasks used to encode the beginning and the end of a plan, respectively. They thus occur only once in each plan and all other tasks are always ordered in between. The effects of the initial task `Init` encode the initial state: there is a contact (James' colleague) that is stored in the cell phone and has a mobile phone number and an email address. Also, the moment the problem gets initialized, the phone has reception and W-LAN connection. The precondition of the task `Goal` encodes the desired goal state. In our example problem, any valid decomposition of the initial plan solves the given problem without the need to explicitly satisfy a goal state.

The initial plan $P_{\text{init}} = \langle TE_{\text{init}}, \prec_{\text{init}}, VC_{\text{init}}, CL_{\text{init}} \rangle$ is formalized as follows:

$$\begin{aligned}
TE_{\text{init}} &= \{l_{\text{init}} : \texttt{Init}(), l_5 : \texttt{ObtainPicture(?pic1)}, \\
&\qquad l_6 : \texttt{SendPicture(?pic2)}, l_{\text{goal}} : \texttt{Goal}()\} \\
\prec_{\text{init}} &= \{(l_{\text{init}}, l_{\text{goal}}), (l_{\text{init}}, l_5), (l_{\text{init}}, l_6), (l_5, l_{\text{goal}}), (l_6, l_{\text{goal}})\} \\
VC_{\text{init}} &= \{\texttt{?pic1} \doteq \texttt{?pic2}\} \\
CL_{\text{init}} &= \{\langle l_5, \texttt{Stored(?pic2)}, l_6 \rangle\}
\end{aligned}$$

3. Plan Generation

In order to find a solution to the problem π, our hybrid planning algorithm has to refine the initial plan into a solution plan P_{sol}.

3.1. Algorithm

Our planning procedure picks a plan P from a candidate set, called *fringe*, which contains all plans that are yet to be examined. Initially, this is only the initial plan, contained in the planning problem. If P constitutes a solution the algorithm returns this plan and terminates. Otherwise, it identifies the problems with P and tries to resolve them. These problems (in the following called *flaws*) explicitly indicate, why this plan does not meet the solution criteria. For example, the second solution criterion states that all tasks of a solution plan have to be primitive. Thus, if a plan P contains abstract tasks, P raises a flaw of type *abstract task* for every abstract task in P. Obviously, the only possibility to resolve such a flaw is to select and apply an appropriate decomposition method $m \in \texttt{M}$, thus replacing the abstract task by the task network specified by m. The procedure of (1) selecting a plan, (2) identifying its flaws, and (3) applying all possible flaw-resolving modifications thereby generating a set of successor plans is repeated until a solution has been found or it has been proven that none exists. This kind of planning procedure is also referred to as refinement planning [1, 19], because each modification application specializes and hence refines the current plan.

A *flaw* is a syntactical structure that references all plan elements that are involved in the violation of a solution criterion. Then, a *flaw class* is the set of all possible flaws of a specific type and is used to relate types of flaws to appropriate types of modifications (see our previous work [20] for formal definitions). For example, the initial plan P_{init} raises (amongst others) two flaws of the flaw class $\mathcal{F}_{\texttt{AbstractTask}}$ which points to the abstract tasks `ObtainPicture` and `SendPicture`.

Table 2: This table lists the solution criteria, their corresponding flaw classes and modifications that can resolve those flaws. Solution criterion (1) is not associated with a flaw class, because any refinement algorithm automatically satisfies it.

Solution Criterion	Flaw Class	Modification Class
(1)	–	–
(2)	$\mathcal{F}_{\texttt{AbstractTask}}$	$\mathcal{M}_{\texttt{ExpandTask}}$
(3)	$\mathcal{F}_{\texttt{OpenPrecondition}}$	$\mathcal{M}_{\texttt{AddCausalLink}}$, $\mathcal{M}_{\texttt{ExpandTask}}$, $\mathcal{M}_{\texttt{InsertTask}}$
(4)	$\mathcal{F}_{\texttt{CausalThreat}}$	$\mathcal{M}_{\texttt{ExpandTask}}$, $\mathcal{M}_{\texttt{AddOrdering}}$, $\mathcal{M}_{\texttt{BindVariable}}$
(5)	$\mathcal{F}_{\texttt{InconsistentOrdering}}$	–
(6)	$\mathcal{F}_{\texttt{UnboundVariable}}$	$\mathcal{M}_{\texttt{BindVariable}}$

In order to resolve a particular flaw f that is detected in a plan P, a *modification* m is applied to P. This results in a modified or refined plan, which is free of f. A modification consists of plan elements to remove from the current plan and plan elements to insert [20]. Note that the application of m might however introduce new flaws into the plan. In our example, a modification that resolves the abstract-task-flaw `SendPicture` would remove this task together with the causal link that points to it. The task would be replaced by an implementing task network. That is, it gets replaced by a task network P for which there is a method $\langle \texttt{SendPicture(?pic2)}, VC, P \rangle \in \texttt{M}$. The causal link would also be replaced, since its consumer would not be the abstract task `SendPicture` anymore, but a more primitive task contained in the plan P.

A *modification class* is the set of all possible modifications of a specific type and is used to relate types of flaws to appropriate types of modifications. For instance, each modification decomposing an abstract task using its associated method belongs to the modification class $\mathcal{M}_{\texttt{ExpandTask}}$. Examples for other modification classes are $\mathcal{M}_{\texttt{InsertTask}}$ and $\mathcal{M}_{\texttt{AddCausalLink}}$, which consist of modifications inserting primitive and abstract tasks and establishing causal links between plan steps, respectively. A complete list of modification classes, as well as the flaw classes they can resolve, is given in Table 2.

As already mentioned, there is a relationship between flaw and modification classes. For example, a flaw of the class $\mathcal{F}_{\texttt{AbstractTask}}$ can only be resolved by a modification of the class $\mathcal{M}_{\texttt{ExpandTask}}$. Thus, only certain types of modifications can be used to address a specific flaw. While this allows for an efficient and distributed calculation of flaws and modifications [21], we will, for the sake of readability, only present a simplified algorithm that calculates and addresses all flaws in one step.

Our procedure is depicted in Algorithm 1; it performs search as long as there are plans in the fringe that can possibly be refined into a solution (line 1). The decision, which plan from the fringe to examine next is made by a function $\mathbf{f}^{PlanSel}$, which we call *plan selection strategy* (line 2). It implicitly defines the

Algorithm 1: Our hybrid planning search procedure.

```
Input  : The candidate set Fringe = $\{P_{\text{init}}\}$.
Output: A solution plan or fail.

while Fringe $\neq \emptyset$ do                                                        1
    $P \leftarrow \mathbf{f}^{PlanSel}(\text{Fringe})$                                   2
    $F \leftarrow \mathbf{f}^{FlawDet}(P)$                                               3
    if $F = \emptyset$ then return $P$                                                   4
    Fringe $\leftarrow (\text{Fringe} \setminus \{P\}) \cup \{\, \mathbf{app}(m, P) \mid m \in \mathbf{f}^{ModGen}(F, P) \,\}$   5
return fail                                                                              6
```

order in which the algorithm explores the space of all possible refinements of the initial plan. If the search process cannot exhaust this space completely, it depends on this function which parts will remain unexplored. $\mathbf{f}^{PlanSel}$ is also the means to introduce optimizations w.r.t. efficiency of search and quality of solution into the search process. Many different plan selection strategies have been described and evaluated in our previous work [15, 22, 20].

After P, the next plan to refine, has been chosen, all flaws are detected by the flaw detection function $\mathbf{f}^{FlawDet}$ and are stored into the set F (line 3). If P does not contain any flaws, it is considered a solution to the given problem and returned (line 4). Otherwise, the fringe is updated by removing the plan currently under consideration and by inserting its successors (line 5). To this end, the modification generation function $\mathbf{f}^{ModGen}$ computes for each detected flaw all possible modifications that resolve it. Note that there are some flaws, e. g., temporal ordering inconsistencies, that persist since they can never be resolved. Let f be such a flaw. We set $\mathbf{f}^{ModGen}(F, P) = \emptyset$ if $f \in F$ in order to discard plans containing these kinds of flaws. The calculated modifications are applied to P (by means of the function **app**) and the resulting plans are inserted into the fringe. If the fringe is empty, i. e., if there is no plan left that can possibly be refined into a solution, the algorithm leaves its main loop and returns ***fail*** (line 6). Figure 8 visualizes this procedure in the context of plan repair.

3.2. Properties of the Algorithm

In the following, we present some theoretical properties of our algorithm and sketch their proofs.

Property 1 (Correctness)**.** *Our algorithm is correct, i. e., if it returns a plan, it is a solution to the given planning problem.*

Obviously, the correctness depends on the completeness of the flaw detection and modification generation functions: every possible violation of a solution criterion must be associated with a flaw, and for every flaw all possible modifications for resolving it must be detected. If both criteria hold – which they do in our framework and implementation [15] – our claim follows by definition.

Property 2 (Completeness)**.** *If there exists a solution to a planning problem, our algorithm finds a solution, provided an appropriate plan selection strategy is chosen.*

This property is related to the fact that we perform search in an infinitely large space of plans, which may be caused by possibly recursive method applications and repeated task insertion, respectively: some tasks may be inserted, either via HTN decomposition methods or via POCL task insertion, arbitrarily often. The former is responsible for the semi-completeness of HTN planning [23], whereas the latter is a general issue of partial order planners like ours [24, 25].

Hence, a search strategy (i. e., the plan selection function $\mathbf{f}^{PlanSel}$) can "get lost" in this infinitely large search space. However, there are search strategies that guarantee to find a solution (if one exists) like the uninformed breadth-first search. Whereas informed search strategies [22, 20] do not guarantee completeness in general, in most practical scenarios they do find a solution while being much more efficient than uninformed search.

Property 3 (Termination)**.** *If no solution exists, our algorithm does not always terminate. If there is a solution, termination depends on the choice of the plan selection strategy.*

Our first claim is easy to see, since for proving the non-existence of a solution the fringe must become empty eventually. But the potential search space is infinite while only a few plans get discarded from the fringe due to unresolvable flaws; hence, termination cannot be guaranteed in cases where no solution exists. We want to emphasize that this is just a theoretical result but not of much importance for our purposes of assisting human users. This is, since we generally assume the existence of a solution and are mainly interested in finding a trustworthy, user-adapted solution quickly, rather than proving that none exists.

Our second claim follows directly from the completeness property, as the algorithm terminates as soon as a solution is encountered.

3.3. Example

Initially, the fringe contains only the initial plan P_{init}, depicted in Figure 5. The flaw detection function $\mathbf{f}^{FlawDet}$ raises three flaws: Two flaws of the class $\mathcal{F}_{\texttt{AbstractTask}}$, one for each of the two initial abstract tasks, and one flaw of the class $\mathcal{F}_{\texttt{UnboundVariable}}$, because the variable `?pic1` is not yet bound to a constant. The possible modifications detected by the modification detection function $\mathbf{f}^{ModGen}$ are applied, thereby generating a set of successor plans. At the time our planner has generated a solution, the abstract task `ObtainPicture` has been substituted by the tasks `EnterCameraMode` and `TakePicture`, and the abstract task `SendPicture` has been substituted by a plan which sends the picture via MMS. Note that in our domain model there are two methods that specify how to decompose the abstract task `SendPicture`. One method uses an MMS to send a picture, another one sends it by email. Our solution depicted in Figure 6 used the former since we assume that the assistance component makes use of a user profile in which the personal user preferences tell the system that

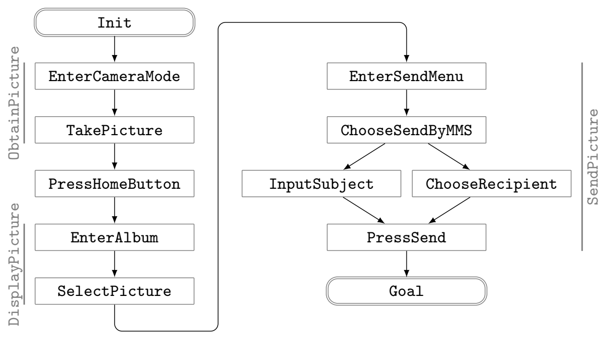

Figure 6: A solution plan to the problem of sending a picture to a contact from the address book of the smart phone. The annotations on the side describe by which abstract task the primitive tasks were introduced into the plan. The abstract tasks `ObtainPicture` and `SendPicture` were already present in the initial plan. The abstract task `DisplayPicture` and the primitive task `PressHomeButton` were introduced into the plan in order to close open preconditions of their succeeding tasks.

James has never used the email functionality before, whereas the task of sending MMS messages is quite familiar to him. Because the task `EnterSendMenu` has the precondition of the respective picture being selected, our planning system inserts the abstract task `DisplayPicture`, which in turn gets decomposed into a more primitive plan. After all flaws have been resolved, a plan has been obtained that is free of flaws and that actually is a decomposition of the initial plan.

Thus, the assistance component of James' cell phone created a solution to the problem π of taking a picture and sending it to a colleague. The solution (cf. Figure 6) is presented to James, who begins executing it.

4. Plan Repair

One basic assumption underlying our planning approach discussed so far is that the only way in which the environment changes is through actions executed by the user. Since we want to offer advanced plan-based user assistance and have to take the human user's very dynamic environment into account, this assumption is too restrictive.

James begins to follow the provided step-by-step instructions. He is able to take a picture of the notes and begins composing an MMS message to his colleague. James then leaves the room and heads for the next talk in a different room. But while doing so, the reception signal of the mobile phone gets weaker and finally completely fades.

A plan generated by the assistance component can be invalidated by exceptional events occurring at plan execution time, such as the fading reception in our example. In this section, we will present an extension to our formalism that meets the need to deal with such unforeseeable developments.

One possibility to cope with unexpected environmental changes is to restart planning from scratch, using the modified situation as the new initial state. Dealing with the problem in this way has the advantage that no modification of the planning algorithm is needed. *Plan repair*, on the other hand, tries to adapt a previously generated solution to a new situation. While plan repair is in terms of computational complexity not easier than replanning in the general case [26], the main advantage of plan repair is that it allows for more *plan stability.* Plan stability is a measure of the similarity between the original solution and the solution adapted to the new situation [27]. The goal is to present alternative plans with only minimal deviations from the original plan, as there are several reasons why plan stability is important.

First, overcoming a plan failure by generating a new solution from scratch may result in a completely different plan, which most likely appears implausible to the user and may lead to major confusion. Preserving as much of the original plan as possible instead, appears much more appropriate and helps to gain and/or preserve a user's trust. Suppose our example solution P_{sol} does not end after James has sent his message but contains further tasks, then repairing the plan will produce a plan close to the original one, while planning from scratch would only produce *some* plan.

Second, solutions to real world planning problems generally tend to be large. In case of a failure, many unexecuted parts of the plan may be unaffected. It is thus not adequate and sometimes even infeasible to create a new plan just to address a single execution failure.

Third, requests for resources or requests to third parties may have already been carried out and undoing them might lead to unfavorable effects: if something goes wrong, say, after James has already bought a flight ticket for his trip home, he is unlikely to accept a new solution that proposes to travel by train.

We will thus use a plan repair approach to deal with unpredictable environmental changes. While there exist several different plan repair approaches for classical *state-based* planning, there are hardly any for *hierarchical* planning [28]. In the following, we present our plan repair procedure for *hybrid* planning, based on our previous work [29].

Before describing the plan repair process, we will show how failures are represented in the planning domain model. In our formalism described in Section 2, the only way states can change is by the execution of actions; the formalism does not foresee external influences that cause those changes. The task of the repair procedure on the other hand is to produce a solution that compensates an unexpected failure. As a consequence, a well-founded approach to plan repair requires the means to explicitly represent failures. Unexpected changes to the environment, sometimes called *events* in the literature [1, page 5f.], can be interpreted as a kind of action themselves if we regard the environment as a second actor in the domain. These actions can of course not be used by the planner to construct plans, because they represent uncontrollable environmental developments. Instead, these so-called *processes* are inserted into the repair plan to represent execution failures. Processes invert the truth value of literals in state descriptions: the failing cell phone reception in our example is modeled

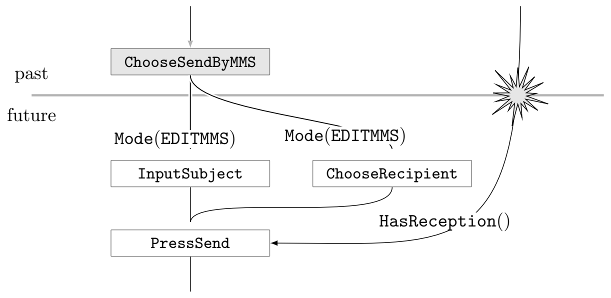

Figure 7: An example of how the execution monitor is used to monitor the evolution of the environment. The horizontal line identifies the *execution horizon*, i. e., the point in time up to which the plan has been executed. Gray nodes are tasks marked as executed by the monitor. If a failure is detected, the causal links crossing the execution horizon are candidates for being failure-affected, because their properties are needed for the execution of a future action. The causal link establishing the precondition `HasReception` for the task `PressSend` is failure-affected, as illustrated by the failure symbol on the right.

by a process $p_{\texttt{HasReception}}$ with precondition `HasReception` and postcondition $\neg$`HasReception`. In this way, we are able to represent all imaginable changes to the environment as a sequence of processes.

As a means to detect plan failures, we assume the existence of an *execution monitor*, which keeps track of both the execution of the actions in the plan and the evolution of the environment. When unexpected changes in the environment are discovered, it identifies which parts of the plan, if any, can no longer be executed as intended. This includes unexpected results of action execution, properties required as preconditions of future actions that do not hold as expected, and so on. The monitor maps these findings on the plan data structure and annotates the problematic parts as *failure-affected*. In our running example, the relation `HasReception` is a property required to enable a future action, namely the task `PressSend`. By the time James arrives at the room of the next talk, `HasReception` becomes false, and the execution monitor can identify the causal link establishing the precondition `HasReception` for `PressSend` as failure-affected. This situation is depicted in Figure 7. The figure also makes clear that the failure assessment is not limited to the actions immediately following the failed plan fragment; the causal structure of the plan allows us to infer and anticipate causal breakdowns for actions to be executed far in the future. We can correctly detect that the future action `PressSend` cannot be executed as intended, even if several actions (`InputSubject`, `ChooseRecipient`) lie in between.

Compared to our plan generation process presented in Algorithm 1, there is slightly more to the plan repair process, because it has to take into account that some parts of the plan are already executed and are therefore *non-retractable planning decisions*. Every executed plan element needs to occur in the repaired plan as well: James cannot go back on what he has already done. Therefore, the basic idea behind our plan repair approach is to (1) make sure that the repaired plan coincides with the failed plan on the executed parts, (2) include the exceptional environmental processes, and (3) control repair plan generation such that the new plan resembles the unexecuted parts of the original plan as well, if possible. Note that we have to include the executed parts of the original solution in the repaired plan. This is because we want it to be a solution to the original planning problem, which might include abstract tasks whose implementations are only executed in parts when the failure is detected. Thus, the solution must contain a valid decomposition of all abstract tasks included in the planning problem, cf. criterion (1) of the solution definition in Section 2.4. To make sure that executed parts of the original solution are contained unaltered in the new solution, we derive a template from the executed parts of the original solution and use it to control the generation of the repair plan. This template is represented using so-called *obligations*: for every executed plan element (like `ChooseSendByMMS` in our example), we introduce a corresponding obligation into the plan template, requiring its existence in the repair plan, including ordering and causal information. To satisfy obligations, corresponding plan elements have to be assigned to them during repair.

4.1. Problems and Solutions

Next, we will formally define plan repair problems and their corresponding solutions. For the repair problem definition $\pi^r = \langle D^r, P^r_{\text{init}} \rangle$, we first augment the domain model $D = \langle \mathtt{L}, \mathtt{T}, \mathtt{M} \rangle$ by adding the processes, resulting in $D^r = \langle \mathtt{L}, \mathtt{T}, \mathtt{M}, \mathtt{Pr} \rangle$, where $\mathtt{Pr}$ is a set of processes as follows: for every relation, we introduce two processes that invert the truth value of the respective relation from *true* to *false* and vice versa. Note that we do not simply add the processes to the set of available tasks $\mathtt{T}$, because the system cannot actively use them to construct plans. The initial repair plan $P^r_{\text{init}} = \langle \mathit{TE}_{\text{init}}, \prec_{\text{init}}, \mathit{VC}_{\text{init}}, \mathit{CL}_{\text{init}}, O_{\text{init}} \rangle$ is constructed by extending the initial plan of the original planning problem $P_{\text{init}} = \langle \mathit{TE}_{\text{init}}, \prec_{\text{init}}, \mathit{VC}_{\text{init}}, \mathit{CL}_{\text{init}} \rangle$ with the obligations O_{init} computed from the failed solution plan P_{fail} by the execution monitor. The obligations in O_{init} have two sources. Firstly, O_{init} contains an obligation for every executed plan element (tasks, causal links, etc.) of P_{fail}. Secondly, execution failures have to be represented. Execution failures are caused by unexpected changes in the environment and may thus falsify the annotated condition of causal links. These failure-affected causal links result in appropriate obligations for processes, i. e., let $\langle l_a, \varphi, l_b \rangle$ be a failure-affected causal link, then O_{init} contains the following obligations o:

- o_a requires the existence of the task expression $te_a = l_a : t_a$,
- o_φ requires the existence of a process p_φ with precondition φ and post-condition $\neg\varphi$, encoding the environmental change,
- and finally $o_{CL} = \langle o_a, \varphi, o_\varphi \rangle$ requires the existence of a causal link between te_a and the process p_φ.

This definition is based on the assumption that all initial goals persist and that the underlying domain model is still adequate and stable. This is the case for our example, neither has anything changed that would make the domain model itself invalid nor has James changed his mind about his goals. We can thus state a plan repair problem that contains obligations up to and including the task `ChooseSendByMMS`, plus obligations as defined above for the process $p_{\texttt{HasReception}}$.

An obligation-extended plan $P^r_{\text{sol}} = \langle TE_{\text{sol}}, \prec_{\text{sol}}, VC_{\text{sol}}, CL_{\text{sol}}, O_{\text{sol}}\rangle$ is a solution to a plan repair problem $\pi^r = \langle D^r, P^r_{\text{init}}\rangle$ if and only if the following conditions hold:

- P^r_{sol} fulfills the planning solution criteria of Section 2.4. This implies in particular that P^r_{sol} is an executable solution to the original problem, if we regard the processes added via obligations as additional tasks.
- The obligations in P^r_{sol} are *satisfied.* A set of obligations O is satisfied w.r.t. a plan P^r, if every obligation in O is assigned to a plan element in P^r such that the ordering is consistent. This ensures that the non-retractable decisions of the failed plan are respected and that the anomaly of the environment is considered.
- For all task expressions te_a and te_b, if O contains an obligation for te_b and $\text{te}_a \prec^*_{\text{CL}} \text{te}_b$, then O also contains an obligation for te_a. This property ensures that no executed plan step can occur after an unexecuted one.

4.2. Algorithm

We need to integrate the repair mechanism into the general hybrid planning framework of Section 3 so that it can solve plan repair problems. To achieve that, we extend the flaw detection function $\mathbf{f}^{FlawDet}$ and modification generation function $\mathbf{f}^{ModGen}$ such that they detect and address unsatisfied obligations. This enables the hybrid planning Algorithm 1 to treat obligations transparently, i. e., no modification to Algorithm 1 is necessary.

Our repair procedure presented in Algorithm 2 starts at the failed original solution plan P_{fail}. This plan is extended with appropriate obligations as stated in the repair problem definition (line 1). To obtain a new solution, we make use of the previously explored plan space. This is in contrast to local search approaches that take the failed plan as-is and try to repair it by locally adding and removing tasks compensating for the failure [27, 30]. We examine the sequence of modifications *ModSeq* our planning system applied to obtain P_{fail} from the initial plan P_{init}. Plan elements are introduced by modifications, so we can determine modifications related to failure-affected plan elements. This yields a partition of *ModSeq* into two subsequences: $ModSeq_{\text{fail}}$, the modifications related to failure-affected plan elements, and $ModSeq_{\text{good}} = ModSeq \setminus ModSeq_{\text{fail}}$ (line 2), the modifications unrelated to failure-affected plan elements. Note that sometimes, the application of a modification depends on the prior application of a modification related to failure-affected plan elements. While such

Algorithm 2: The repair algorithm for hybrid planning.

```
Input  : Repair problem $\pi^r = \langle D^r, P^r_{\text{init}} \rangle$, the failed solution plan $P_{\text{fail}}$,
         and the modification sequence $\mathit{ModSeq}$ leading from $P_{\text{init}}$ to $P_{\text{fail}}$.
Output: A repaired plan or fail
$P^r_{\text{fail}} \leftarrow$ addObligations($P_{\text{fail}}, \pi^r$)                                   1
$\mathit{ModSeq}_{\text{good}}, \mathit{ModSeq}_{\text{fail}} \leftarrow$ partition($\mathit{ModSeq}$)              2
$P^r_{\text{retr}} \leftarrow$ retract($P^r_{\text{fail}}, \mathit{ModSeq}$)                                  3
$P^r_{\text{safe}} \leftarrow$ reapply($P^r_{\text{retr}}, \mathit{ModSeq}_{\text{good}}$)                         4
$P^r_{\text{current}} \leftarrow P^r_{\text{safe}}$                                             5
repeat                                                          6
    $P^r_{\text{new}} \leftarrow$ generatePlan($P^r_{\text{current}}, \pi^r$)      // call Algorithm 1        7
    if $P^r_{\text{new}} \neq$ fail then                                            8
        return $P^r_{\text{new}}$                                                9
    else                                                        10
        // retract last modification
        $P^r_{\text{current}} \leftarrow$ retractOne($P^r_{\text{current}}$,popLast($\mathit{ModSeq}_{\text{good}}$))          11
until popLast was applied to the empty sequence $\mathit{ModSeq}_{\text{good}}$           12
return fail                                                     13
```

modifications are not directly related to failure-affected plan elements, we include them in $\mathit{ModSeq}_{\text{fail}}$. In our example, the modification that introduced the causal link establishing the precondition `HasReception` for the task `PressSend` is therefore included in $\mathit{ModSeq}_{\text{fail}}$, while the modification that added the task `InputSubject` to the plan is contained in $\mathit{ModSeq}_{\text{good}}$, because the execution of `InputSubject` does not depend on having reception. We proceed by retracting modifications in reverse order of application until we encounter P^r_{retr}, the first plan without failure-affected elements (line 3). In doing so, we have retracted all modifications in $\mathit{ModSeq}_{\text{fail}}$ and possibly some in $\mathit{ModSeq}_{\text{good}}$. The retracted modifications which are contained in $\mathit{ModSeq}_{\text{good}}$ are reapplied in line 4, yielding P^r_{safe} without involving combinatorial search. Thus, **reapply** does not have to reapply all modifications in $\mathit{ModSeq}_{\text{good}}$, but only those below P^r_{retr}. Our basic search procedure (Algorithm 1) is then started on P^r_{current}, which is initially P^r_{safe} (line 5), and attempts to refine it into a solution (line 7). If a solution P^r_{new} is found, it is returned by the algorithm. If **generatePlan** returns ***fail***, the last modification is retracted from P^r_{current} in line 11, because there might be a solution for the new initial plan P^r_{current} to which the retracted modification was not applied. If there are no more modifications to retract (line 12), the main loop terminates and the plan repair returns ***fail*** in line 11, stating that no solution exists. Figure 8 visualizes the plan space traversed during this process.

Our algorithm preserves all modifications unaffected by the failure, thereby producing a plan close to the original solution. To further improve the similarity to the original solution, a least-discrepancy heuristic can be used to guide the actual planning in line 7. These measures accomplish minimal invasiveness.

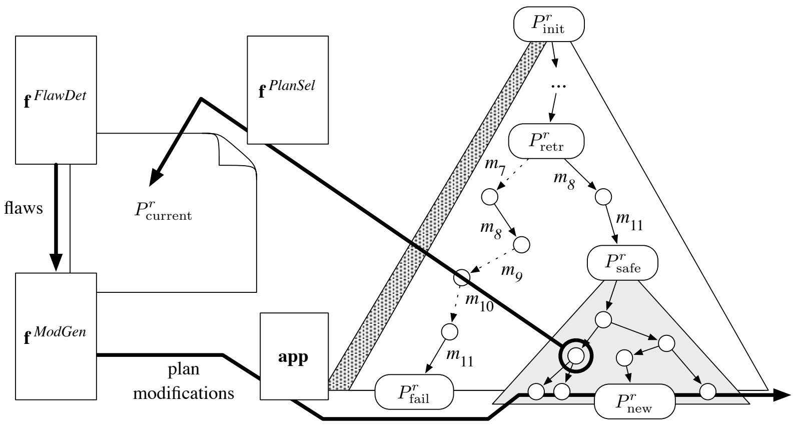

Figure 8: The left hand side sketches the hybrid planning procedure (Algorithm 1). First, a plan is selected from the fringe. For this plan, all flaws are detected, for which in turn all possible modifications are generated. These modifications are applied and the resulting plans are finally inserted into the search space depicted on the right side: It shows a visualization of the plan space traversed during the plan repair process. Modifications m_i are denoted as arrows. Dashed arrows (m_7, m_9, m_{10}) represent modifications related to failure-affected plan elements, continuous arrows (m_8, m_{11}) are unaffected modifications. Unaffected modifications are reapplied to obtain P^r_{safe}. Triangles below a plan represent the plan space reachable from that plan.

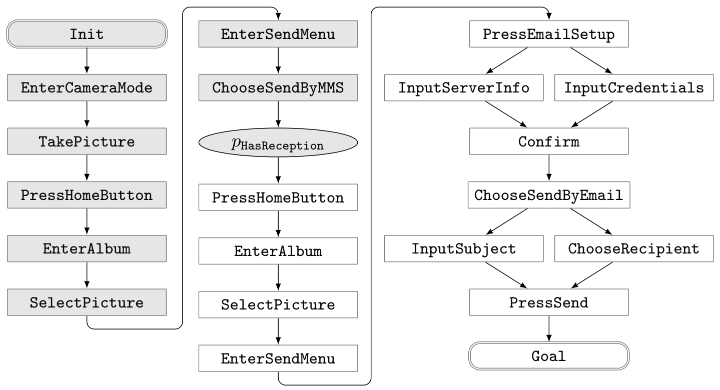

Figure 9: The result of the repair process in our example: James is instructed to set up an email account and send his picture via email using W-LAN. Nodes with a gray background denote executed tasks. $p_{\texttt{HasReception}}$ is the inserted process, representing the anomaly of the environment. Arrows denote ordering constraints. Obligations and causal links are omitted for the sake of readability.

The assistance component of the cell phone tells James that the initially generated plan cannot be carried out due to fading reception. Instead, the assistant proposes an alternative course of action: as the phone has W-LAN functionality and there is a wireless network in range (the conference offers free W-LAN access for participants), James is instructed to send his picture via email. Because he has never sent an email before from his new cell phone, he is also guided through the process of setting up an email account on the device first.

The originally generated solution (cf. Figure 6) failed due to the fading reception (cf. Figure 7). It is hence extended with obligations for the executed parts including the task `ChooseSendByMMS` and the process $p_{\texttt{HasReception}}$. The algorithm then retracts the modifications that introduced failure-affected elements. Afterwards, the resulting plan is refined to a new solution by adding tasks implementing an alternative method to send the picture. Figure 9 shows the result of the repair process in our example. It is apparent that the new solution is also a solution to the original planning problem, though it contains the process $p_{\texttt{HasReception}}$ to reflect the anomaly of the environment. Plan repair thus enables James to reach his goals despite the changing conditions.

4.3. Properties of the Algorithm

Because the main plan generation algorithm is called within the plan repair procedure (line 7), the repair algorithm's properties are basically inherited from the plan generation algorithm (cf. Section 3.2). We will thus not discuss the properties in detail but summarize them by the following statement:

Property 4 (Correctness, Completeness, and Termination)**.** *The properties Correctness, Completeness and Termination of the repair algorithm are the same as the corresponding properties of the planning algorithm as long as it terminates eventually for each problem P^r_{current} it is called with except for the root plan P^r_{init}.*

Property 3 states that the main planning procedure does not necessarily terminate in the general case, neither in cases where there is a solution, nor in cases where there is none. For our procedure, this is a problem because Algorithm 1 is called with another repair problem as soon as the previous run terminated. Pragmatically, this problem can be avoided by simply forcing termination, for instance by returning ***fail*** if no solution could be found within a predefined time limit. Termination may not be forced for the root plan P^r_{init}, as correctness and completeness would be sacrificed.

5. Plan Explanation

This section addresses the challenge of communicating an automatically generated plan to the user. This is not restricted to the way in which a solution is presented, e. g., graphically or via natural speech. More important and fundamental is the problem of getting the user to accept and understand a produced solution. We want to provide as much information as is needed for the user to trust the system and be convinced that the generated plan is appropriate to solve the given problem. Our formalism offers the hierarchical and causal structuring of planning problems and their solutions. In our opinion, this constitutes a fundamental part when aiming to reflect how humans view and think about plans, and as such is essential for conveying not only the generated plans themselves but also making the reasoning behind them transparent. While this might very well help a user to better grasp the problem and its structure, we do not focus on the didactic effect of explanation, i. e., on learning to solve the problem without technical assistance.

The issue of explaining plans that were produced by AI planning systems has not yet been addressed by the planning community. In the following paragraphs, we present our view of what some important aspects of plan explanation might be, and point out how our formalism can be used to address them. They encompass the problems of presenting plans, providing abstract views on plans, giving detailed operating instructions to perform primitive tasks, answering queries about intermediate states, justifying plan elements, and reasoning about alternative solutions.

Presenting Plans. After the planning phase is finished and a solution plan has been generated, the found solution must be presented to the user. In general there exist many ways of achieving this. We discuss the employment of the two main modalities vision and speech.

Let us go back to the moment right after James has told the assistance application that he wants to take a photo and send it to his colleague. The integrated planner has worked out a solution and now needs to present it to James. Because he will need to operate the touch screen it cannot be used for graphical output. Thus, the system uses the speaker of the phone to instruct James for his task: "Touch the camera symbol at the bottom of the screen."

Of the two modalities mentioned, the graphical option seems to be the more powerful version in general. By presenting a plan as a directed graph of plan steps that are connected by ordering constraints, it is possible to convey an impression of the complete solution at once. Figure 6 depicts the first solution of our example in such a way. By looking at a graphical presentation, one is able to assess the extent of the plan and the temporal orderings as well as possible concurrences of actions. But the expressiveness of imagery can also lead to overburdening a user with information. Especially in everyday applications, this is not a desired property.

Modalities that are less prone to the danger of providing too much information at once are text and speech. Using them to communicate a linearized version of a plan to the user results in something that resembles an ordinary manual and enables the user to execute the generated plan one step at a time. Depending on the situation, this might be the preferred mode of presentation. Additionally, the environment might prohibit the usage of a certain modality (e. g., vision when driving a car, audio in loud places).

The presentation of plans, which we consider as the most basic form of plan explanation, is also the one that has been researched the most. A recent

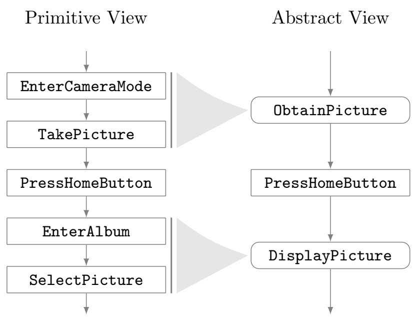

Figure 10: Primitive plan steps are summarized by the abstract tasks they were decomposed from. This results in an easier to understand representation.

publication of our group presents a working prototype system that uses a text-to-speech server and an annotated domain model to verbally present and explain plans that stem from our hybrid planning formalism [31]. A three-dimensional presentation of hierarchical plans has been investigated by Kundu et al. [32]. Another interesting approach using an ontology to describe output modalities and planning concepts to generate a visualization has been pursued by Lino et al. [33].

Operating Instructions. We are claiming that our planning formalism is well-suited to generate instructions for human users. However, the result of the planning process is a partially ordered network of plan steps, and instructions of the form `InputCredentials`(`ACC2`) are not easy to interpret. Additionally, the primitive tasks in a plan do not necessarily correspond to the basic actions in a one-to-one way. For example entering a subject for a message might require operating the virtual touch keyboard of the smart phone which in turn is a non-trivial series of actions. Nevertheless, the planning model abstracts from this fact and models this sequence of actions as the single primitive task `InputSubject`.

Therefore, the provision of detailed instructions for tasks is an important aspect. In order to enable non-expert users to execute the automatically generated plans, the system must be able to provide actual operating instructions. To this end, Bidot et al. [31] developed an extension of the domain model syntax of our planning formalism that enables the annotation of utterances for describing tasks via natural speech. An example in our domain could be the linking of the task `InputSubject` to the instruction "Use the virtual keyboard to enter a subject for the message into the box labeled *Subject*.". This approach could also be extended to cover multimodal communication by providing more sophisticated markups.

Abstract Views on Plans. A plan can become rather large and conveying a comprehensive view of it is difficult. It is thus reasonable to look for ways to reduce the information contained in the presentation. One way of achieving this can be the abstraction of parts of the plan.

James is a bit annoyed because the assistance application seems to think he does not know how to take a picture. He feels very able to achieve this without help, so he demands that he is given an overview of the upcoming instructions. The screen switches to a graphical high-level presentation of the remaining plan. James can extract that he is intended to take the picture, open it again by using the album function and then transmit it by MMS. He decides to try the first two steps on his own and tells the system to turn off the instructions for a moment.

The plans generated by our planning system possess an intrinsic hierarchical structure; some plan steps have been inserted as a decomposition of an abstract task. Thus, replacing the result of a decomposition by its corresponding abstract task when presenting the plan is an obvious step. Figure 10 gives an example for an abstract view on a fragment of the solution to our example problem. The presented fragment deals with taking a picture and then displaying it in order to open the menu for sending it. Focusing on the upper half of the figure, the solution to taking a photo consists of two sub tasks: entering the camera mode of the smart phone (`EnterCameraMode`) and then triggering the camera (`TakePicture`). These two plan steps together constitute a method to implement the abstract task `ObtainPicture`. By using abstractions, the understandability of a hierarchical plan can be improved.

When reducing information in such a way, it is important to regulate the degree of abstraction applied. In more complex settings, a solution will contain deeper hierarchies of decomposition. It is an open question how a reasonable level of abstraction can be automatically determined.

When presenting abstract views on plans in an assistance application like the one in our example, information about the user should be exploited extensively. For example, an expert user might be able to execute a common task without further instructions. Thus, a task like this can be presented in an abstract way for such a user while being presented on a primitive level for novices. By using information about the level of complexity of tasks, a system can decide which abstraction to choose depending on the involved task, taking into account the general level of expertise of the user. When using a more fine-grained user model that provides an estimate of the user's abilities concerning particular tasks, it becomes possible to provide a level of abstraction that varies for each task depending on the confidence of the particular user with that task. This would be an improvement over the use of mere user classes like *expert* or *novice.*

We can summarize that the adaptation of the level of abstraction is an important tool when communicating plans, but the exact details of an ergonomic and intuitive regulation remain subject to further research.

Queries About Intermediate States. It is very important to have information about the state of the domain while executing a plan. Especially when applying planning technology to real world domains, focusing only on reaching the goal is questionable. The execution of a plan may have undesirable side-effects and it is important that those can be inspected.

Using the camera on his new phone has proven to be much easier than James had expected. The manufacturer seems to be on the right track concerning usability. However, the last step of sending the picture as an MMS remains to be undertaken. And both because he is of a very cautious nature (some people might call him paranoid) and because he wants to test the boundaries of the assistance application he asks whether the picture will still be available on the phone after having been transmitted by MMS.

In the case of totally ordered operator-based planning, obtaining the state of the domain during or after the execution of a plan can be achieved in a canonical way. Applying the plan steps one after another to the initial state, one obtains the states during the plan execution. But since we are dealing with partially ordered plans, we cannot completely determine the exact way in which the plan will be executed unless it is linearized before execution. For our example application, having the solution linearized by the system is a valid option and by doing so we can take advantage of the easier access to intermediate states. But in general, a plan produced by a hybrid planning system retains its partially ordered nature. And thus when asking questions about an intermediate or a final state of such a plan, we have to consider every linearization. This can be $n!$ linearizations for a plan consisting of n plan steps and no ordering constraints between them.

Dean and Boddy [34] have shown that the computational complexity of reasoning about states for a constellation of partially ordered events can range from polynomial time to NP-hardness; but their partially ordered events are merely conditionally executable. In contrast, all plan steps that exist in a solution to a hybrid planning problem will have their preconditions satisfied, and thus will be executable unconditionally. Therefore, we expect that the reasoning task about final and intermediate states during the execution of solutions of partially ordered plans is feasible.

The fact that we are dealing with partially ordered plans is also reflected in questions and their answers. A question like “Does property X hold after performing task T?” translates into “Does property X hold after performing task T in every linearization?”. Accordingly, answers to questions about the state of partially ordered plans reflect this as well. The response “Sometimes.” might occur very often if the question is not specific enough.

Justifying Plan Elements. Another category of questions that can be addressed are those that aim at the justification of certain plan elements. A user may be puzzled when confronted with a solution to a complex problem and may be inclined to ask why a certain plan step is necessary.

Before James was ready to hit the final send button the reception waned and the assistance application suggested a fixed solution to his problem. Since the system created the impression of a mature and well-thought-out piece of software, James simply followed the instructions guiding him along the way of *sending the picture by using the available W-LAN. At the point where he is instructed to enter the user name and password of his email account, doubts begin to emerge again and he asks why this action is a necessary step.*

Before providing an answer to such a question we have to think about what can be considered a justification. The only commitment by the user is the stipulation of the initial planning problem. We assume that this problem correctly reflects the goals that the user wants to achieve. Thus, every justification ultimately has to root there.

Taking our example, the system could respond to James' question with the immediate reason "Entering a user name and password is necessary for setting up an email account.". But the user might not be satisfied with this answer and think "Why do I have to set up an email account? I just want to send a picture to my colleague.". To close the chain of justification, the system might have better answered "Entering a user name and password is necessary for setting up an email account, which in turn is needed to send the picture by email to your colleague.".

The refinement-based hybrid planning formalism provides us exactly with this *chain of justification*. Every deviation away from the initial task network is conducted by a modification that addresses a particular flaw, and therefore can be justified by the necessity of resolving that flaw. For example, a plan step consisting of the task `InputCredentials` may have been introduced to address a flaw of type $\mathcal{F}_{\texttt{AbstractTask}}$ that was associated with a plan step `SetUpEmail-Account`. So by keeping the modification sequence that led to a solution, we can extract from it the chains of justifications that explain the necessity of certain plan elements.

To further improve on these justification chains, the system might skip certain elements inside the chain if it can be assumed that the user is able to fill the gap by using his or her own reasoning capabilities. It is also imaginable that the user model is keeping track of given explanations and this information can be used to determine which parts of a justification chain can be simplified or skipped completely. For example, if the user was already given the reason of the existence of the plan step `SetUpEmailAccount`, the chain of justification of the plan step `InputCredentials` can end there, assuming that the user has accepted the necessity of `SetUpEmailAccount`.

Reasoning about Alternative Solutions. The last paragraph has dealt with justifying the existence of plan components that constitute a found solution. This means we can substantiate the contribution of a certain element of the plan to solving the problem. This does *not* mean that this element must be present in every possible solution.

James does now understand that he has to set up his email account in order to send the picture by email. But he is not very willing to do so because he's slow with the touch keyboard. To make matters worse, the next talk has already begun and he doesn't want to spend any time on unnecessary things. Therefore, *he queries the assistance component for an alternative way to get his problem solved.*

During the plan generation process, the system makes decisions about how to resolve certain flaws that are present at an intermediate planning step. These decisions have to be supported by a user that is supposed to execute the resulting plan. A reasonable question is thus one about the reasons behind those decisions.

The hybrid planning formalism we are promoting can potentially lead to an infinite number of solutions. Depending on the domain model, there might exist a task that can be inserted arbitrarily many times without breaking the solution. For our example this might be using the home button, which resets the device to a defined state. The problem the planning system faces is to find and pick one or more of these plans for presentation to the user. If the plan space is finite and can be completely explored by the search process, answering questions about the existence of alternative solutions is trivial, giving either a positive or negative answer. Asking for a justification of why a particular solution has been preferred by the system might be more difficult and is depending on the exact way user preferences are handled. It then boils down to justifying the choice of the preference system. In the case of infinitely many solutions or a search space that cannot be completely explored because of its size, the issue becomes more and more problematic. Absolute answers are not obtainable and the system must decide how and where to spend additional computational power in order to answer questions to the best extent possible.

Of all presented aspects of the explanation of plans, the question about alternatives is the most complicated one. This is simply because it has to deal with the largest space that contributes to the answers, the complete search space. Justifying plan elements, as we are interpreting it, has to deal only with one given path through the search space – the one going from the initial problem to the solution plan. Questions about plan state are concerned only with the solution plan itself and instructions for actions only address one single plan step and maybe its parameters.

6. Conclusion

We have shown that our approach of hybrid planning enables the realization of complex cognitive capabilities of technical systems.

Automated reasoning techniques including the generation, repair, and explanation of plans can serve as core components of systems that provide advanced user assistance. In our example, such a system supports a user while he operates a mobile communication device.

User instructions are provided based on plans of action that are synthesized by the hybrid planning system. If the execution of a certain action fails due to some unexpected change of the environment, for example, the system is able to help the user out of that situation by initiating a plan repair process. The resulting plan overcomes the failure situation and is stable by exhibiting only those deviations from the original plan that are indispensable. Finally, plan explanation can be provided based on an analysis of the knowledge-rich plan structures generated by the planner as well as of the planning process itself.

Future work is devoted to developing such an explanation generator and provide it as an additional reasoning component to the envisaged assistance system. Furthermore, the question of how the system actually becomes aware of a user's intentions will be addressed. Besides the option to consult an underlying user model and compute the most likely next goal based on deposited typical user plans and the observed operation history, like it is done in plan recognition, also a speech dialogue with the user may be initiated. While in our setting, assistance is provided only in case of need when operating the device, additional modes like those where the user can explicitly ask for support or the system acts as a tutor, i. e., a proactive electronic instruction manual, would be useful enhancements of the portfolio of assistance functionalities.

Plan-based assistance is however not restricted to the support of users setting up or operating technical devices. There is a broad spectrum of prospective applications including personal organizers to support (elderly) people in their everyday decision making and medical assistance systems that accompany patients in rehabilitation processes, for example. For these kinds of applications, two additional features of our hybrid planning approach are essential and will have to be exploited: the generation of *individualized user plans* and the generation of (partially) *abstract solutions* to a given planning problem.

Acknowledgement

This work was partly accomplished within the Transregional Collaborative Research Centre SFB/TRR 62 "Companion-Technology for Cognitive Technical Systems", funded by the German Research Foundation (DFG).

References

[1] D. S. Nau, M. Ghallab, P. Traverso, Automated Planning: Theory & Practice, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[2] R. E. Fikes, N. J. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208.

[3] A. L. Blum, M. L. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1-2) (1997) 281–300.

[4] D. McDermott, The 1998 AI planning systems competition, AI Magazine 21 (2) (2000) 35–55.

[5] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (2001) 5–33.

[6] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.

[7] M. Helmert, The fast downward planning system, Journal of Artificial Intelligence Research 26 (2006) 191–246.

[8] D. McAllester, D. Rosenblitt, Systematic nonlinear planning, in: Proceedings of the Ninth National Conference on Artificial Intelligence, 1991, pp. 634–639.

[9] J. S. Penberthy, D. S. Weld, UCPOP: A sound, complete, partial order planner for ADL, in: Proceedings of the third International Conference on Knowledge Representation and Reasoning, 1992, pp. 103–114.

[10] K. Erol, J. Hendler, D. S. Nau, UMCP: A sound and complete procedure for hierarchical task-network planning, in: Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS 1994), 1994, pp. 249–254.

[11] Q. Yang, Intelligent Planning. A Decomposition and Abstraction Based Approach, Springer, 1998.

[12] S. Kambhampati, A. Mali, B. Srivastava, Hybrid planning for partially hierarchical domains, in: Proceedings of the 15th National Conference on Artificial Intelligence, American Association for Artificial Intelligence (AAAI Press), 1998, pp. 882–888.

[13] L. A. Castillo, J. Fernández-Olivares, A. González, A hybrid hierarchical/operator-based planning approach for the design of control programs, in: ECAI Workshop on Planning and Configuration: New results in planning, scheduling and design, 2000, pp. 1–10.

[14] S. Biundo, B. Schattenberg, From abstract crisis to concrete relief (a preliminary report on combining state abstraction and HTN planning), in: Proceedings of the 6th European Conference on Planning (ECP 2001), Springer-Verlag, 2001, pp. 157–168.

[15] B. Schattenberg, Hybrid planning & scheduling, Ph.D. thesis, Ulm University, Germany (2009).

[16] S. Andrews, B. Kettler, K. Erol, J. A. Hendler, UM Translog: A planning domain for the development and benchmarking of planning systems, Tech. Rep. CS-TR-3487, University of Maryland (1995).

[17] T. A. Estlin, S. A. Chien, X. Wang, An argument for a hybrid HTN/operator-based approach to planning, in: Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning, 1997, pp. 182–194.

[18] L. A. Castillo, J. Fernández-Olivares, A. González, On the adequacy of hierarchical planning characteristics for real-world problem solving, in: Proceedings of VI European Conference of Planning, 2001.

[19] S. Kambhampati, Refinement planning as a unifying framework for plan synthesis, AI Magazine 18 (2) (1997) 67–98.

[20] B. Schattenberg, J. Bidot, S. Biundo, On the construction and evaluation of flexible plan-refinement strategies, in: J. Hertzberg, M. Beetz, R. Englert (Eds.), Advances in Artificial Intelligence, Proceedings of the 30th German Conference on Artificial Intelligence (KI 2007), Vol. 4667 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2007, pp. 367–381.

[21] B. Schattenberg, S. Balzer, S. Biundo, Knowledge-based middleware as an architecture for planning and scheduling systems, in: D. Long, S. F. Smith, D. Borrajo, L. McCluskey (Eds.), Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), American Association for Artificial Intelligence (AAAI Press), Ambleside, The English Lake District, UK, 2006, pp. 422–425.

[22] B. Schattenberg, A. Weigl, S. Biundo, Hybrid planning using flexible strategies, in: U. Furbach (Ed.), Advances in Artificial Intelligence, Proceedings of the 28th German Conference on Artificial Intelligence (KI 2005), Vol. 3698, Springer-Verlag, 2005, pp. 249–263.

[23] K. Erol, J. Hendler, D. S. Nau, HTN planning: Complexity and expressivity, in: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994), 1994, pp. 1123–1128.

[24] S. Kambhampati, Admissible pruning strategies based on plan minimality for plan-space planning, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1995), 1995, pp. 1627–1633.

[25] D. E. Smith, M. A. Peot, Suspending recursion in causal-link planning, in: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS 1996), American Association for Artificial Intelligence (AAAI Press), 1996, pp. 182–190.

[26] B. Nebel, J. Köhler, Plan reuse versus plan generation: a theoretical and empirical analysis, Artificial Intelligence 76 (1-2) (1995) 427–454.

[27] M. Fox, A. Gerevini, D. Long, I. Serina, Plan stability: Replanning versus plan repair, in: Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006), American Association for Artificial Intelligence (AAAI Press), 2006, pp. 212–221.

[28] I. Warfield, C. Hogg, S. Lee-Urban, H. Muñoz-Avila, Adaptation of hierarchical task network plans, in: Proceedings of the Twentieth Flairs International Conference (FLAIRS 2007), American Association for Artificial Intelligence (AAAI Press), 2007, pp. 429–434.

[29] J. Bidot, B. Schattenberg, S. Biundo, Plan repair in hybrid planning, in: A. Dengel, K. Berns, T. Breuel, F. Bomarius, T. R. Roth-Berghofer (Eds.), Advances in Artificial Intelligence, Proceedings of the 31st German Conference on Artificial Intelligence (KI 2008), Vol. 5243 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2008, pp. 169–176.

[30] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in LPG, Journal of Artificial Intelligence Research 20 (1) (2003) 239–290.

[31] J. Bidot, S. Biundo, T. Heinroth, W. Minker, F. Nothdurft, B. Schattenberg, Verbal plan explanations for hybrid planning, in: 24th MKWI related PuK-Workshop: Planung/Scheduling und Konfigurieren/Entwerfen, 2010, pp. 455–456, full article available at `http://webdoc.sub.gwdg.de/univerlag/2010/mkwi/03_anwendungen/planen_scheduling/06_verbal_plan_explanations_for_hybrid_plannings.pdf`.

[32] K. Kundu, C. Sessions, M. des Jardins, P. Rheingans, Three-dimensional visualization of hierarchical task network plans, in: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, 2002.

[33] N. Q. Lino, A. Tate, Y.-H. J. Chen-Burger, Semantic support for visualisation in collaborative AI planning, in: The International Conference on Automated Planning and Scheduling (ICAPS 2005), workshop on The Role of Ontologies in AI Planning and Scheduling, Montery, California, USA, 2005, pp. 37–43.

[34] T. Dean, M. Boddy, Reasoning about partially ordered events, Artificial Intelligence 36 (3) (1988) 375–399.