30th International Conference on
Automated Planning and Scheduling

October 19 – 30, 2020, ~~Nancy (France)~~ online!

HPlan 2020

Proceedings of the 3rd ICAPS Workshop on

Hierarchical Planning

Organizing Committee

Pascal Bercher	The Australian National University, Canberra, Australia
Daniel Höller	Saarland University, Saarbrücken, Germany
Roman Barták	Charles University, Prague, Czech Republic
Ron Alford	The MITRE Corporation, McLean, Virginia, USA

Program Committee

Ron Alford	The MITRE Corporation, McLean, Virginia, USA
Roman Barták	Charles University, Prague, Czech Republic
Gregor Behnke	University of Freiburg, Germany
Pascal Bercher	The Australian National University, Canberra, Australia
Susanne Biundo	Ulm University, Germany
Rafael C. Cardoso	The University of Manchester, UK
Kutluhan Erol	İzmir University of Economics, Turkey
Christopher Geib	SIFT, LLC, Minneapolis, MN, USA
Florian Geißer	The Australian National University, Canberra, Australia
Patrik Haslum	The Australian National University, Canberra, Australia
Daniel Höller	Saarland University, Saarbrücken, Germany
Dana Nau	University of Maryland, College Park, Maryland, USA
Conny Olz	Ulm University, Germany
Sunandita Patra	University of Maryland, College Park, Maryland, USA
Mak Roberts	Naval Research Laboratory, Washington, DC, USA
Vikas Shivashankar	Amazon Robotics, North Reading, Massachusetts, USA
Julia Wichlacz	Saarland University, Saarbrücken, Germany
Zhanhao Xiao	School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China

Special thanks goes to the HPlan program committee, who worked through a tremendous amount of COVID-induced chaos and delay to provide each submission with high-quality feedback.

Preface

The motivation for using hierarchical planning formalisms is manifold. It ranges from an explicit and predefined guidance of the plan generation process and the ability to represent complex problem solving and behavior patterns to the option of having different abstraction layers when communicating with a human user or when planning cooperatively. This led to numerous hierarchical formalisms and systems. Hierarchies induce fundamental differences from classical, non-hierarchical planning, creating distinct computational properties and requiring separate algorithms for plan generation, plan verification, plan repair, and practical applications. Many issues required to tackle these – or further – problems in hierarchical planning are still unexplored.

With this workshop, we bring together scientists working on many aspects of hierarchical planning to exchange ideas and foster cooperation.

Like in previous years, a range of topics is addressed in the submitted papers. Half of the accepted papers deal with heuristic search. Two of these introduce novel HTN planning heuristics: One bases on the concept of landmarks, and the other on an encoding of delete-relaxed HTN problems using (Integer) Linear Programming (LPs and ILPs). The third paper introduces Monte Carlo Tree search into HTN planning. One paper attempts to establish a well-founded link between HTN planning, HTN plan recognition, and natural language processing via the introduction of a new planning model based on formal grammars. Another paper deals with repairing failed plans, while the last one formalizes (German) legal opinions in terms of HTN planning knowledge as a basis for tool support for students, lawyers, and judges.

Due to the ongoing COVID restrictions, ICAPS – and thus the HPlan workshop – will go online. Just like the main conference and (most, if not all of) the other workshops we will use gather.town to meet virtually. To establish some of the "ICAPS family feeling" despite being online, we decided to only have poster presentations (except for 5 minute poster teaser talks, but these will be pre-recorded, streamed, and without discussion/questions, as these will be moved to the poster presentation). In those poster sessions, each presenter has designated area within the virtual workshop room, where he/she waits. Once he/she gets approached, he/she shares his/her screen and starts the (group) conversation. So everybody can just join in at an arbitrary time or even approach other (groups of) people during the workshop within the room – thus giving back a bit of the real-world workshop and conference experience.

Pascal, Daniel, Roman, and Ron,
HPlan Workshop Organizers,
October 2020

Table of Contents

Proceedings of the 3^{rd} ICAPS Workshop on Hierarchical Planning (HPlan 2020)

Formalising German Legal Opinions as Planning

Gregor Behnke
University of Freiburg
behnkeg@informatik.uni-freiburg.de

Abstract

A legal opinion in the German legal system is a formal piece of writing that investigates whether a given statement of law is true or not given a description of a specific case. Writing these opinions is the central element of German legal education, but is supported only by basic IT technologies, such as text-based search engines. Formalising legal thoughts would enable the creation of various tools that support students, lawyers, and judges in correctly applying the law.

German legal opinions are interesting are interesting from a research perspective as they follow a strictly formalised structure and method of argumentation. In practice, these opinions can (often) be seen as a thorough application of so-called schemata. A schemata provides a fixed way to check whether a specific assertion of law holds or not by providing sub-assertions to check and a rule on how these results should be combined. In essence, these schemata therefore describe a hierarchical (but potentially recursive) structure on legal terms and properties.

We propose a formalisation of these schemata in terms of Hierarchical Task Network (HTN) planning. The modelled domain will describe the application of the law on the specifics of a given case s.t. the resulting plan and its decompositional structure will constitute the structure of a legal opinion on the case.

1 Introduction

Almost any action has some connection to the legal system we are living in. Either by means of civil law (governing the relationship between peoples and between people and objects), public, or criminal law (dealing with the connection between the state and its citizens). Despite its pervasiveness, the legal argumentation is still a relatively manual task. Especially when learning to apply the law, there is almost no automated support – apart from e.g. search engines.

Applying the law is a multi-step process. First, the facts relevant to the case in question must be gathered. Second, these facts are associated with legal concepts – a process called subsumption. For example one may have to decide whether the letter written by a vendor is a "firm summons to pay" (which is required for a notice of default). Third, we have to reason about legal consequence, e.g. who has an obligation to pay. The first two steps deal with the fuzzy and uncertain aspects and are thus more suited for sub-symbolic techniques like machine learning.

The third step may however be viewed as "purely logical". At least for the anglo-american common law system, this is not the case, as legal reasoning here hugely depends on matching prior cases against the facts of the case at hand – which again is a fuzzy process. Thus research for common law systems often deals with reasoning on the basis of precedents (Atkinson and Bench-Capon 2019). In civil law systems, the application of the law depends only on the written law. The constant jurisprudence of the highest courts may still play a role in the interpretation of the written law, but is not as dominant as is the common law. As such, there is a body of work dealing with the formalisation of legal reasoning, e.g. as Deontic Logic (Jones and Sergot 1992), as Answer Set Programming (Aravanis, Demiris, and Peppas 2018), as PROLOG rules (Satoh et al. 2011), as ontologies (Palmirani and Governatori 2018), or as argumentation (Marshall 1989; Prakken and Sartor 2015).

How reasoning about the consequences of given facts under the law is performed depends on the legal system. Germany's legal system uses opinions to derive consequences, which follow a highly formalistic and logic-driven style of argumentation (Kischel 2019, p. 417ff). Notably, there exist pre-determined ways and means for determining whether a statement X follows from the specifics of a given case, so-called schemata. For this, the schema sets out a sequence of other statements that have to be checked and a rule on how to combine them.

In this paper, we present a formalisation of these schemata in terms of planning. With this formalisation we are (1) able to derive the truth of a statement given an appropriately modelled fact (at least to some degree) automatically and (2) lay the foundation for future automated assistance of legal scholars, layers, and students. Notably, the plan generated for a given fact will correspond to the structure of the opinion for checking it. We could, for example, use our formalisation as the basis for teaching student specific concepts of law – by developing opinions in cooperation with

them. Note that we do not aim at fully writing an opinion, we rather aim at representing the structure and argumentation contained in the opinion. Any verbalisation would be out of scope of a conference paper and we consider it future work.

We start by giving a brief introduction into Legal Opinions and their structure and will then introduce our running example that we will consider throughout the paper. We then introduce the concept of HTN planning. Thereafter we discuss how the structure of opinions and their schemata can be represented in terms of HTN planning problems.

2 Legal Opinions and German Civil Law

Legal reasoning is – at least in Germany – most often contained in written so-called opinions. An opinion is a structured collection of arguments what show why a legal result (e.g. a title or a right) is entailed by the law and the specifics of a given case. For example, an opinion may ask whether a person B has the obligation to pay a given amount of money to a claimant A. Writing opinions is the central element of legal training at German universities. Almost all university exams require students to write an opinion on a given case. Further, the First State Exam (the first part of the German bar exam) consists of six written tests, each asking to write a legal opinion. The strict adherence to structured and logical arguments required for opinions sets legal education in Germany apart from the system used in many other countries. Learning to write these opinions is however quite hard for most students. Thus, supporting the process of learning to write opinions with AI technologies may be a fruitful endeavour.

Argumentation in a legal opinion follows a strict style of writing, called the opinion-style. In order to determine the truth of a statement of law, one has to perform the four steps of a syllogism, called the legal syllogism (German: "Justizsyllogismus"):

1. the premise,
2. the definition,
3. the subsumption, and
4. the result.

The premise states that a certain statement (about an obligation, a title, a right, or any other property or legal connection between two persons or a person and an object) might be true. Then one defines the criteria under which the statement is true, citing the relevant law. Within the subsumption, the specifics of the case are mapped to the criteria set out in the definition. Usually, the subsumption is the part of an opinion that requires case-dependent argumentation. As the subsumption is based on the verbal description of the case, it is a highly fuzzy process. Lastly, if the requirements set out in the definition are met, one concludes that the premise is indeed true. If the definition again contains assertions that cannot be directly mapped to specifics of the case, as they are e.g. legal terms, the subsumption will contain further legal syllogisms with these assertions as their premises. These syllogisms are nested recursively.

We do not consider how subsumptions are made, as we want to focus on the logical structure of opinions. Notably, the AI techniques that should be used for performing or assisting subsumptions are quite different from the ones we use, i.e. they might be based on machine learning, natural language processing, and text mining. Instead, we assume that all possible subsumptions of the case have already been made. We assume that we are presented with a given case in terms of a set of logical atoms (instantiated ground predicates) which refer to the most basic concepts of the law. Note that this is not a restriction, but solely a clean separation between the logical and the fuzzy part of the opinion. If we, in the future, are to apply this modelling we of cause have to deal with extracting such a logical description based on non-logical inputs. This may either be done using machine learning techniques, or in collaboration with the user.

Example Case Whenever possible, we will use an example case to illustrate the principles of our formalisation of opinions in terms of planning problems. On June 21st K[1] asked V whether he can buy a set of white floor tiles from V – for 500 EUR. On June 23rd, V agreed. V delivered the tiles to K three days later. K then installed the tiles in his bathroom. Four months later, K noticed that the tiles changed colour from white to greyish. An investigation revealed that this was caused by a production error. K tells V that he wants new tiles delivered to him. Neither K nor V could have noticed the production error before the tiles were installed. Removing the installed tiles and installing the new tiles will cost 400 EUR.

In this case there are two questions: (1) is V required to provide K with new tiles? (2) does V have to pay (additionally) 400 EUR. We will elaborate on the specifics of the case and the answers to the two questions throughout the paper.

3 HTN Planning

In this paper, we will use (totally-ordered) HTN planning (Ghallab, Nau, and Traverso 2004; Erol, Hendler, and Nau 1996; Geier and Bercher 2011) to model the structure of German legal opinions. HTN planning distinguishes two types of tasks: abstract tasks and primitive actions. Both are described via a name and a list of parameter variables, each associated with a type. For example, `(foo ?bar ?baz)` denotes a task named `foo` with parameters `?bar` and `?baz`. We use the syntax of PDDL (McDermott 2000) to denote tasks. Variable names always start with a question mark. A state is described via ground atoms of first order logic, i.e. predicates with constants as their arguments. A state is any subset s of these atoms. As in classical planning, primitive actions in HTN planning carry a state transition semantics, defined via their preconditions $prec$ and effects eff. $prec$ may be any function-free first order formula referring to the variables that are parameters of $prec$'s actions. A (ground) action a is executable in a state s, iff $s \models prec$. The effect eff of an action consists of two sets of atoms add and del which again may refer to the parameters of a. If a is executed in s, it results in the state $(s \setminus del) \cup add$. The execution of sequences of states is defined inductively.

[1] Persons are customarily abbreviated by upper-case letters.

Abstract tasks represent more complex courses of action and do not carry a (direct) state-transition semantics. Instead, their semantics is defined via so-called decomposition methods $m = (t, tn)$. Here, t is the task the method decomposes and tn is a task network – for totally-ordered HTN planning this is a sequence of other tasks, both primitive and abstract. tn can also be an empty sequence. Applying such a method means to replace an occurrence of t with tn. The objective in HTN planning is given in terms of an initial abstract task t_I. We now maintain a sequence π of tasks and initialise it with t_I. We repeatedly apply decomposition methods to tasks in π, until π contains only primitive actions. This derived plan π is a solution to the HTN planning problem if it is executable in the given initial state s_I.

Additionally, many HTN planners (e.g. SHOP (Nau et al. 1999)) allow for *method preconditions*. Such a precondition $prec$ associated with a method m restricts the application of m to cases where the state prior to the first task originating from m satisfies $prec$. Since we consider only totally-ordered models, this is equivalent to the following restriction. Consider a current sequence of tasks $\pi = t_1 \dots t_{i-1} t_i t_{i+1} \dots t_n$. Then a method precondition for a method m applicable to t_i must hold in the state between t_{i-1} and t_i.

4 Representing the Structure of Opinions

As stated before, we assume that the specifics of the case we are to consider are already converted into a set of atoms. In our modelling, these atoms form the initial state s_I. In our example case, this includes e.g. the facts `(hasCondition whiteTiles beingGrey)`, `(usualCondition whiteTiles beingWhite)`, and `(handover V K whiteTiles June26)`. All of them model (relevant) aspects of the given case.

A legal opinion in its entirety determines whether a given premise holds or not. Throughout the opinion, sub-opinions may discuss different other premises, each asking whether a specific statement of law holds. Thus, we have to represent the notion of a premise in terms of the concepts of HTN planning. A natural way to do so is to model premises as tasks. This way, e.g. the initial abstract task t_I will represent the premise of the whole opinion. In our example case, this would be either `(obligationToProvide K V tiles)` or `(obligationToPay K V 400EUR)`. In an opinion, we have to state the definition pertaining to the premise P. Next, we either have to perform a subsumption or have to start inner opinions that determine the truth of statements made in P's definition.

If we only have to perform a subsumption, the premise P only depends on the *most basic* concepts of the law, which have to be inferred from the textual description of the case. As stated before, we do not deal with the intricacies of extracting knowledge from the case in this paper, but only with modelling a legal opinion based on it. As such, we assume that the facts of the case are already fully modelled as ground first order statements and can thus be checked in a primitive action's precondition. Thus, any premise P that only requires a "pure" subsumption will be modelled as a primitive action who's preconditions will check the necessary facts.

If the premise's definition refers to *non-basic* concepts, we have to introduce inner opinions. We do this via modelling such a premise as an abstract task A and introduce the inner opinions via a method m applicable to A. m's task network contains a task for every premise that needs to be checked in order to be able to determine the truth of the main premise A.

The derivation of a plan π will correspond to the structure of an opinion for the given case. To extract the structure of the opinion, we simply follow the decomposition methods applied for obtaining π. We start out with an opinion containing the premise represented by the initial abstract task t_I. When looking at the method applied to t_I, we know which definition and sub-premises should be inserted into the opinion. Similarly, if a decomposition yields a primitive action, the opinion will contain a subsumption.

5 Schemata for Inner Opinions

We have just stated that decomposition methods will introduce the necessary inner opinions. This assertion is somewhat vague and needs clarification. The central question at this point – and of this paper – is which criteria need to be checked in order to be able to establish the truth of a given statement of law. For that – at least in civil law systems – we have to turn to the written law. Most legal norms describe at their core an implication where a set of premises are mapped to a consequence. There are other types of legal norm, e.g. describing the intention of the legislator (e.g. § 1 of Germany's Nuclear Law: "The purpose of this law is (1.) to end the use of nuclear energy for commercial purposes in an orderly fashion ...") or describe broad abstract statements (e.g. Article 1 of Germany's Basic Law: "Human dignity shall be inviolable."). These types of statements are not used for directly determining whether a given statement of law is true, but (for (1.)) for interpreting under-specified terms in the law. How such interpretations may be considered within a formalised version of the law is out of scope of this paper and may be considered in future work. Thus we are interested in norms that provide as their consequence the statement A – our current premise represented by an abstract task. Of course there can be multiple norms providing the same consequence, and there are often norms that provide exceptions, counter-exceptions, counter-counter-exceptions, and so on for a given assertion. These are known to be quite difficult to handle in a general fashion; see for negation in legal reasoning e.g. (Kowalski 1989).

When writing an opinion the complexities of these logical structures are often simplified by – to some degree – standardised structures for checking whether a given assertion is true or not (Kischel 2019, p. 419f.). Instead of solely applying the word of the law, in many practical cases there are established so-called *schemata* that outline how a specific assertion should be checked, in which order the individual premises should be checked, which exceptions and counter-exceptions should be considered, and in which order. There are even whole books that summarise these schemata for different areas of the law (Maties and Winkler 2018). This apparent "standardisation" serves (in the authors' opinion) the purpose of conformity of expectations: the reader of an

opinion knows what the writer wants to do next and why the author does or does not discuss certain issues at each point while reading the opinion. Schemata also ease a student's understanding of the law, as they are able to "follow them" and by that will obtain the correct result in applying the law. For that however, they first have to understand their meaning, learn them, and be able to extract them from the written law.

Naturally there are situations when applying the law, where no schemata are available, i.e. where the meaning of the law must be interpreted manually. How this is done is a quite complex topic, as it e.g. includes argumentation of the intent of the legislator. Formalising this kind of fuzzy area of applying the law is thus difficult and we consider it to be out of scope of our work. We will restrict our modelling to areas of the law where well-established schemata for applying the law exists. The difficulty in practise lies in correctly applying these schemata and extracting the relevant facts from a description of the case. The work in this paper is however still necessary and useful, as it lays the foundation for being able to connect reasoning about the more fuzzy parts of legal argumentation with those that are more standardised.

6 Logical Structure of Schemata

As discussed in the previous section, when modelling what needs to be checked in order to establish the truth of a given statement of law, we use the well-defined schema for it. Such a schema is essentially a list of conditions under which a given statement holds. As an example, consider the requirements for supplementary performance in German Warranty Law. Supplementary performance is the process or act with which a bought object that is defective is repaired or replaced, i.e. the case where the buyer is given e.g. a non-working object and thus has the right to either get the object repaired or replaced. This is exactly the situation we face in question (1) of our example case: K wants to have the tiles replaced.

Supplementary performance is required if (1) vendor and buyer have a valid sales contract, (2) the purchased object has a "defect" (German: "Sachmangel"), (3) this defect was present when the risk passed from the vendor to the buyer (German: "Gefahrübergang") and (4) the rights arising from defects are not precluded ((Maties and Winkler 2018), Nr. 178). If one is to check whether a right to supplementary performance exists, one has to check these four criteria. Further, there are two types of supplementary performance: repair and replacement. In our example case, K wants the tiles to be replaced. The schemata for replacing an object is this case required (1) that a right to supplementary performance exists, (2) K has selected replacement, and (3) no right to withhold performance because of disproportionality ((Maties and Winkler 2018), Nr. 178). When checking whether K has the right to new tiles, the top-most structure of the opinion is as shown in Fig. 1. In an opinion the conditions just listed always appear in the same fixed order. Further, if one of the conditions is not satisfied, the opinion must discuss all conditions prior to the not satisfied one as well, but not the ones afterwards. If, e.g. the object has no defect under the law, one still has to show that there is a valid sales contract, but can omit discussing the passing of the risk and so on.

However, not all schemata describe conjunctions of conditions. For example, there are in total seven causes for a defect of an object – detailed in § 434 and § 435 BGB.[2] In order for an object to have a defect under the law, one of these causes suffices, i.e. they form a conjunction. One might think that checking only one – namely the one constituting the defect – might be sufficient in an opinion. This is not the case. The causes for a defect have an implicit and customary order in which they have to be discussed ((Maties and Winkler 2018), Nr. 176 and 177). Notably, if e.g. the fourth cause of a defect is present, but not the first three, an opinion must discuss and reject the first three causes. On the other hand, there are also disjunctions, for which it is not necessary to check all possible causes until the first one is successful. Examples are e.g. causes for the invalidity of declarations and causes for a suspension of the statute of limitations. Here, it is sufficient to check the single cause that will lead to the desired result. Lastly, there are also disjunctions for which all conditions must be checked fully, i.e. one has to check any possible criterion causing a statement to become true, irrespective of the truth of the other criteria. One notable example for this is the crime of causing bodily harm ("Körperverletzung"): it requires that the perpetrator to either "physically assault" someone or to cause "damages to health". In an opinion, both criteria must be checked – even if the first one is already fulfilled – while one of the two being fulfilled is sufficient.

When modelling schemata in an HTN planning problem, we will thus distinguish four types of logical connectors:

1. conjunction,
2. ordered disjunction,
3. free disjunction, and
4. complete disjunction.

Next, we have to consider that it is not only sufficient for an opinion to check whether a given statement of law holds. In many cases it is also necessary to check whether the contrary is true, i.e. that a given statement does not hold. We have already seen two such cases above, namely the conditions (4) no preclusion and (3) no right to withhold. As a further example, a cause for a defect is that the vendor has not provided a manual and the buyer did not successfully assemble the object (§ 434 II 2 BGB, the so-called IKEA-clause). Similarly, a claim can only be pursued as a general rule, if the statute of limitations has not expired. Note that these negations can be nested in complex cases – or if we want to check e.g. the absence of a claim between two persons.

Thus our modelling of schemata has to handle negations. A purely logical approach to negation is not suited as legal opinions require specific structures of argumentation, which are different from a purely logic-based argumentation. Consider the negation of a conjunction, i.e. we want to check that a given statement is not true and the truth of this statement is based on a conjunctive schema. From a logical point-of-view, it is sufficient to find one condition x in the conjunc-

[2] BGB = Bürgerliches Gesetzbuch, Germany's civil code

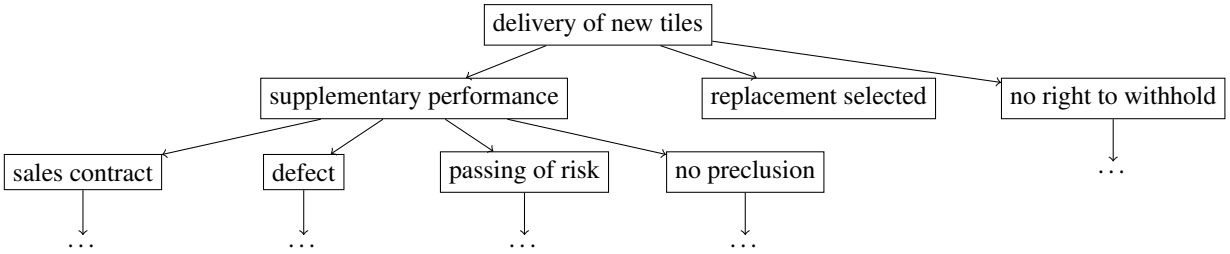


Figure 1: Structure of an opinion for the example case at its top layers.

tion that is not true. However in an opinion, one also has to show that all conditions preceding x are true, i.e. one has to find the *first* failing condition. For all three types of disjunctions, we have no alternative but to show that all its conditions are false.

7 Representing the Logic of Schemata

As we have discussed in Sec. 4, the inner opinions needed to ascertain the truth of a statement A shall be the tasks contained in methods decomposing A. Which assertions we have to check in inner opinions and how their result influences the truth of the overall statement A is determined by A's schema. In our modelling, the method decomposing A will always contain tasks for all conditions $B_1, \ldots, B_n$ contained in the schema. If, in a concrete opinion, a specific condition B_i does not have to be checked, it will be decomposed using an empty decomposition method – denoting that it should not be part of the opinion. Hence, what remains is to provide a mechanism for suitably modelling the logical structure of the schema and its implications on the assertions to be checked in an opinion.

Representing Negation as a Parameter

Next, we note that checking a specific assertion, i.e. an abstract task, may either be done with the objective of proving the assertion or proving the negation of the assertion. Thus it might seem sensible to add a parameter to each abstract task, denoting the mode in which it is to be checked. If we do this though, every task inside a method will need such a parameter. Next, these parameters must be separate variables for each condition in a schema, as if we check a conjunction or the negation of a disjunction, not all conditions are checked with the same objective. Assertions in the methods precondition would have to ensure that the variables are set correctly for each of the subtasks. We don't use such a modelling for three reasons. First, it unnecessarily increases the size of the ground instantiation as the variable determining the objectives for each B_i. Second, such a modelling may make the model harder to solve when using grounded progression search algorithms, which are currently one of the best algorithms for HTN planning (Höller et al. 2018). Since the variables are part of the decomposition method, we have to *guess* which of the conditions $B_1, \ldots, B_n$ will hold when we apply the decomposition method. With the modelling we propose below, we defer this decision until the point where we actually check each of the individual conditions B_i. Third, the rules on determining the conditions' objectives have to be encoded once per method, thus cluttering the model with unnecessary repetitions and making it less readable. With the method that we propose below, these rules are encoded only once in the domain and remain static, e.g. if new material law is added to the domain – thus easing modelling significantly.

Representing Negation as a State Variable

Instead of using parameter variables to denote the objective with which we check each of the conditions $B_1, \ldots, B_n$, we use a *mode* that we set in the state. We model the mode as a predicate `(mode ?m)`. Each abstract task A (i.e. its decomposition) will check the objective set out by the mode which holds in the state directly before A. After A has been checked in the given mode, our modelling sets the current mode back to the mode with which A has been checked. If it is not possible to check the assertion of A in the given mode, it will not be possible to decompose it into an executable plan. This way, we can guarantee that the created opinion is in itself consistent: a valid plan can only be found if the result matches the assertion that was set out in the beginning. Further, it allows us to create planning problems where the result of the opinion is unknown. Here, the initial abstract task is a dummy, that decomposes into the actual premise of the opinion preceded with a primitive action that will either set to mode to check the premise positively or negatively.

In addition to two modes representing objectives, we introduce three additional modes we use for controlling which conditions are checked for negated conjunctions and non-negated disjunctions. We use the following five modes with their respective meaning:

- `yes` – we attempt to show that the condition holds
- `no` – we attempt to show that the condition does not hold
- `indifferent` – we have to show that the condition holds or not, but the result is irrelevant
- `ignore` – we do not have to consider this condition, but do not yet know whether the main statement A is true nor not
- `done` – we do not have to consider this condition, but already know whether the main statement A holds

Any method for A uses its method precondition to determine the current mode via a precondition `(mode ?initMode)`. If `?initMode` is `yes`, we know that we have to check the assertion represented by A positively, if it

is `no` negatively. If the mode is `indifferent`, it does not matter whether we check the statement positively or negatively. These three cases are handled by the same decomposition method. This way we have to model the schema for A only once in the planning domain and eliminate unnecessary redundancy. The method precondition of this method checks that `?initMode` is either `yes`, `no`, or `indifferent`.

If `?initMode` is either `ignore` or `done`, we don't have to check the assertion at all. Thus for these cases, the planning domain contains an empty decomposition method (i.e. one without subtasks) that checks whether `?initMode` is either `ignore` or `done` in its precondition. In total, we create only two decomposition methods for each statement A, one for the case where we actually have to execute A's schema and one where we completely forgo checking A. The first type of methods actually applying A's schema will contain a subtask for each of the conditions $B_1, \ldots, B_n$. Whether any condition B_i will be checked in a concrete opinion is determined by the method applied to it – if the second type of method is applied, it will not be contained in the opinion. This way, we can model the schemata fully inside a single method – without any complicated if-then-else structure in the method.

Methods with Mode-Checking

When checking an assertion A via a method of the first type, we have to set the correct checking-mode before each of the tasks representing the conditions $B_1, \ldots, B_n$ contained in A's schema. The corresponding tasks then have to perform their checks according to the set mode. Once the task has been executed, we have to set the mode for the next task, execute it, and repeat until all conditions of the assertion's schema have been handled. Determining which mode can be used for each of the subtasks depends on multiple factors: the type of the logical connector that the schema uses, the mode of the assertion we are currently checking, and the mode with which the previous subtasks have been checked. Firstly, instead of basing the result on the modes for all previous subtasks, we only consider the last one. Its mode will have aggregated all necessary information about the already checked conditions. For example, if a condition B_i was successfully checked with the mode `yes` in a conjunction, all previous conditions B_j with $j < i$ were also checked successfully with the mode `yes`. Here, the modelling capabilities of HTN planning come into play: The method that has checked the previous condition will have set the mode back to the mode it was "called". Via this mechanic, we know in A's method the mode with which the subtask was called without the need for adding additional variables to the method – which would increase the size of the model's grounding and make the method itself less readable. Secondly, we use the variable `?initMode` to determine the checking mode of A itself. This is – once again – only possible as we are using HTN planning. It allows us to enforce that a given set of actions – those setting the modes – share a common parameter. This essentially forms a kind of a brace-like structure, i.e. a context-free structure, which are not expressible with classical planning, but with HTN planning (Höller et al. 2014).

Thus the mode that should be set of a condition B_i depends on: (1) the logical connector, (2) the overall mode of A, and (3) the mode that was set to B_{i-1}. In Tab. 1 we show how we set the mode for B_i depending on these three inputs. We additionally show how we set the mode initially (the *start mode*). Lastly, we list which mode is required to hold after the last condition B_n has been handled. With this check, we ensure that in the case of a positively checked disjunction one of the conditions was checked with the `yes` mode and for a negatively checked conjunction that one of the conditions was checked with the `no` mode. For the reverse cases (positive conjunction and negative disjunction), the correct modes are already enforced by the mode setting.

To set the modes we use a primitive action `(set-mode ?logic ?initMode)` implementing the state transition function of Tab. 1. Here `?logic` is a variable that can be instantiated with four constants: one for each of the four types of logical connectors. We use actions `(start-mode ?logic ?initMode)` and `(end-mode ?logic ?initMode)` for setting the initial mode, checking the final mode and setting the mode after the last condition back to `?initMode`. We add the `set-mode` action prior to every task B_i in a method and `start-mode` and `end-mode` as the first and last tasks in each method.

The mode-setting in Tab. 1 assumes that the conditions B_i are occurring positively in the schema. For handling negation, we use an additional action `invert`. If executed, it will change the `yes` mode into `no` and vice versa – effectively implementing a negation of a specific check. It will not alter the mode if it is set to anything other than `yes` or `no`. In total, if schema states that A holds if either B, not C, or D hold, its method will contain the following tasks and actions:

1. `(start-mode disj-seq ?m)`
2. `(set-mode disj-seq ?m)`
3. `(B)`
4. `(set-mode disj-seq ?m)`
5. `(invert)`
6. `(C)`
7. `(invert)`
8. `(set-mode disj-seq ?m)`
9. `(D)`
10. `(end-mode disj-seq ?m)`

If we are to use this method in a setting where neither B, nor C, nor D holds, A will hold. The execution and mode-setting in this case is shown in Fig. 2.

8 Modelling

We have modelled (parts of) German Warranty Law using our methodology. Our modelling solves the two questions we set out for our example case in the beginning of the paper.

For this, we also had to model parts of connected areas of civil law, e.g. parts of contracts law, or consumer rights. The domain currently consists of 42 abstract tasks, 84 lifted decomposition methods, and 35 lifted primitive actions. The model is available at https://github.com/galvusdamor/htnlaw. The example case we've outlined in the beginning is solved by an SAT-based HTN planner (Behnke, Höller,

target	last	conjunction	ordered disjunction	free disjunction	complete disjunction
yes	start mode	yes	no	ignore	no
no	start mode	yes	no	no	no
yes	yes	yes	done	done	indifferent
yes	no	–	yes/no	–	yes/no
yes	ignore	–	–	ignore/yes	–
yes	done	–	done	done	–
yes	indifferent	–	–	–	indifferent
no	yes	yes/no	–	–	–
no	no	done	no	no	no
no	done	done	–	–	–
yes	end mode	yes	yes/done	yes/done	yes/indifferent
no	end mode	no/done	no	no	no
indifferent	end mode	yes/no/done	yes/no/done	yes/no/done/ignore	yes/no/indifferent

Table 1: State transitions for modes. Start mode indicates the mode in which checking starts while the last two lines (denoted with "end mode") denotes the states in which checking may end with success. Dashes show transitions that are not allowed.

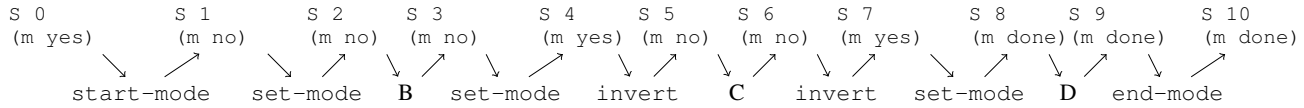


Figure 2: Structure of the evaluation of the assertion: A, if either B, not C, or D. The disjunction is an ordered disjunction. We assume that neither B, C, nor D hold, i.e. A holds because C does not hold. The initial objective is to show that A holds. `S i` denotes that ith state, while `(m x)` indicates the mode `x` set in this state.

and Biundo 2019a; 2018; 2019b) in 13.7 seconds on an Intel i5-4300U.

9 Conclusion and Outlook

In this paper, we showed that the structure of German legal opinions can be formalised in terms of HTN planning. This formalisation represents objectives to be checked as abstract tasks and uses decomposition methods to check whether the assertion holds using the established schemata for doing so. We showed how the various types of logical structures occurring in these schemata can be modelled within decomposition methods by exploiting the high expressiveness of HTN planning. The presented modelling relies on a prior formalisation of the facts of a given case in terms of appropriate first order atoms. In the future, we may use the modelled domain to enable a semi-supervised formalisation of subsumption. We may, in turn use this to assist law students in learning to structure their thoughts and to write opinions themselves. For example, we could verify that the structure of options written by students complies with the schemata using HTN plan verification (Behnke, Höller, and Biundo 2017; Barták, Maillard, and Cardoso 2018). Based upon the detected errors, we could develop techniques to provide useful hints to the students on how to improve their opinions.

References

Aravanis, T.; Demiris, K.; and Peppas, P. 2018. Legal reasoning in answer set programming. In *Proceedings of the 30th International Conference on Tools with Artificial Intelligence (ICTAI 2018)*, 302–306. IEEE Computer Society.

Atkinson, K., and Bench-Capon, T. 2019. Reasoning with legal cases: Analogy or rule application? In *Proceedings of the 17th International Conference on Artificial Intelligence and Law (ICAIL 2019)*. ACM.

Barták, R.; Maillard, A.; and Cardoso, R. C. 2018. Validation of hierarchical plans via parsing of attribute grammars. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2017. This is a solution! (... but is it though?) – Verifying solutions of hierarchical planning problems. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 20–28. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. totSAT – Totally-ordered hierarchical planning through SAT. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, 6110–6118. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7520–7529. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 5500–5508. IJCAI.

Erol, K.; Hendler, J.; and Nau, D. 1996. Complexity results for HTN planning. *Annals of Mathematics and AI* 18(1):69–93.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 1955–1961. AAAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, volume 263, 447–452. IOS Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, B. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 114–122. AAAI Press.

Jones, A., and Sergot, M. 1992. Deontic logic in the representation of law: Towards a methodology. *Artificial Intelligence and Law* 1(1):45–64.

Kischel, U. 2019. *Comparative Law*. Oxford University Press, 1. edition.

Kowalski, R. 1989. The treatment of negation in logic programs for representing legislation. In *Proceedings of the 2nd International Conference on Artificial Intelligence and Law (ICAIL 1989)*, 11–15. ACM.

Marshall, C. 1989. Representing the structure of a legal argument. In *Proceedings of the 2nd International Conference on Artificial Intelligence and Law (ICAIL 1989)*, 121–127. ACM.

Maties, M., and Winkler, K. 2018. *Schemata und Definitionen Zivilrecht: mit Arbeits-, Handels-, Gesellschafts- und Zivilprozessrecht*. C.H.Beck, 1. edition.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*, 968–973.

Palmirani, M., and Governatori, G. 2018. Modelling legal knowledge for GDPR compliance checking. In *Legal Knowledge and Information Systems - JURIX 2018: The Thirty-first Annual Conference*, 101–110.

Prakken, H., and Sartor, G. 2015. Law and logic: A review from an argumentation perspective. *Artif. Intell.* 227:214–245.

Satoh, K.; Asai, K.; Kogawa, T.; Kubota, M.; Nakamura, M.; Nishigai, Y.; Shirakawa, K.; and Takano, C. 2011. Proleg: An implementation of the presupposed ultimate fact theory of japanese civil code by prolog technology. In Onada, T.; Bekki, D.; and McCready, E., eds., *JSAI International Symposium on Artificial Intelligence – New Frontiers in Artificial Intelligence (JSAI 2011)*, 153–164. Springer Berlin Heidelberg.

Proceedings of the 3^{rd} ICAPS Workshop on Hierarchical Planning (HPlan 2020)

Landmark Extraction in HTN Planning

Daniel Höller
Saarland Informatics Campus,
Saarland University,
Saarbrücken, Germany
hoeller@cs.uni-saarland.de

Pascal Bercher
College of Engineering and Computer Science,
The Australian National University,
Canberra, Australia
pascal.bercher@anu.edu.au

Abstract

Landmarks are state features that need to be made true or tasks that need to be contained in every solution of a planning problem. They are a valuable source of information in planning and can be exploited in various ways. Landmarks have been used both in classical and hierarchical planning, but while there is much work in classical planning, the techniques in hierarchical planning are less evolved. In this paper we introduce a novel landmark generation method for Hierarchical Task Network (HTN) planning and show that it is sound and incomplete. We show that every complete approach is as hard as the underlying HTN problem. Since we make relaxations during landmark generation, this means **NP**-hard for our setting (while our approach is in **P**). On a widely used benchmark set, our approach finds more than twice the number of landmarks than the approach from the literature. Though our focus is on landmark *generation*, we show that the newly discovered landmarks bear information beneficial for solvers.

1 Introduction

Two widely used approaches to planning are *classical planning* and *Hierarchical Task Network (HTN) planning*. In classical planning, the environment is described using a set of (propositional) state features that are modified by *actions*, which define valid state transitions. The objective is to find a sequence of actions transforming the initial state of the system into one in which certain goal features hold.

In HTN planning there are two kinds of tasks: actions like in classical planning (also called *primitive tasks*) and *abstract tasks*. The latter are not applicable directly, but are decomposed into other (primitive or abstract) tasks by using *decomposition methods*. The objective in HTN planning is not to fulfill a state-based goal condition, but to find an executable decomposition of given abstract tasks. Since there is (usually) more than one method for an abstract task, the hierarchy implies a second combinatorial problem because a planner has to choose the "right" method for a certain task. This makes HTN planning more expressive (Erol, Hendler, and Nau 1996; Geier and Bercher 2011; Höller et al. 2014; 2016). The decomposition process can be seen as an AND/OR tree (Ghallab, Nau, and Traverso 2004; Kambhampati, Mali, and Srivastava 1998). Starting with the initial task, a planner chooses a single method (i.e. abstract tasks form OR nodes) and has to include all subtasks into the plan (i.e. methods form AND nodes), and so on.

A concept that has been successful especially in classical planning is that of *landmarks* (LMs). LMs are state features (or actions) that are made true (contained) in every solution. It was first used for problem decomposition (Porteous, Sebastia, and Hoffmann 2001; Hoffmann, Porteous, and Sebastia 2004) and later for creating non-admissible (see e.g. Zhu and Givan (2003), Richter, Helmert, and Westphal (2008), and Richter and Westphal (2010)) and admissible heuristics for heuristic search (see e.g. Karpas and Domshlak (2009) or Helmert and Domshlak (2009)). LMs have also been introduced in hierarchical planning. First in form of *task* LMs in hybrid planning (Elkawkagy, Schattenberg, and Biundo 2010) (an extension of HTN planning), later in form of *fact* LMs in HGN planning, a formalism where the hierarchy is defined on *goals*, not on tasks. The former can directly be applied to HTN planning and will be the baseline for our approach. While the latter can apply LM generation techniques from classical planning directly (Shivashankar et al. 2013; 2016a; 2016b; Shivashankar, Alford, and Aha 2017), the presented techniques are not applicable to HTN planning. We summarize landmark-related work in the context of HGN planning in Section 5.

Work on LMs can be divided into two orthogonal categories (Keyder, Richter, and Helmert 2010): *LM utilization* showing how to exploit LM information, and *LM generation*, showing how to find LMs. We focus on the latter.

Based on techniques from classical planning, we introduce a novel approach for LM generation in HTN planning that elegantly combines the extraction of fact, action, and method LMs. It dominates the existing work (finding at least the same LMs). Our approach is sound and incomplete. We further show that every (sound and) complete approach is as hard as the underlying planning problem, i.e., in our setting – delete-effects and ordering-relations of the HTN model are ignored during generation – **NP**-hard (while our approach is in **P**). On a widely used benchmark set we find more than twice the number of LMs than related work. Though our focus is on LM *generation*, we further show that the additional LMs bear valuable information for search guidance.

2 Formal Framework

A classical planning problem P is a tuple (F, A, s_0, g, δ). F is a set of propositional *state features* (or *facts*) that is used to describe the environment. A *state* $s \in 2^F$ is given by those state features that hold in it, all others are supposed to be false. $s_0 \in 2^F$ is called the *initial state* and $g \subseteq F$ is the goal condition. A is a set of *action names*. The functions $\delta = (prec, add, del)$ with $prec, add, del : A \to 2^F$ map action names to a set of state features defining the action's *precondition*, *add effects*, and *delete effects*, respectively. An action a is *applicable in a state* $s \in 2^F$ if and only if its precondition is contained in the current state, $prec(a) \subseteq s$. When a is applicable in s, the state s' resulting from its application is defined as $s' = (s \setminus del(a)) \cup add(a)$. A sequence of actions $(a_1, a_2, \ldots, a_n)$ is applicable in a state s when action a_i with $1 \leq i \leq n$ is applicable in state s_{i-1}, where s_i for $1 \leq i \leq n$ results from applying the sequence up to action i. The state s_i is called the *state resulting from the application*. All states $s \supseteq g$ are called *goal states*. A *plan* (or *solution*) is a sequence of actions applicable in s_0 that results in a goal state.

We now extend classical problems to HTN problems based on the formalism by Geier and Bercher (2011). An *HTN planning problem* $\mathcal{P} = (F, A, C, M, s_0, tn_I, g, \delta)$ extends a classical problem by a decomposition hierarchy on the things to do, the *tasks*. Tasks are divided into *primitive tasks* equal to actions in classical planning, and *abstract tasks* that can not be applied directly but need to be decomposed first. Let A and C[1] be the sets of primitive and abstract tasks, respectively. We assume that their intersection is empty and call the set of all task names $N = A \cup C$.

Tasks are organized in *task networks*. A task network is a triple $tn = (T, \prec, \alpha)$, where T is a set of identifiers (ids), $\prec$ a strict partial order on the ids, and α a mapping from ids to actual tasks $\alpha : T \to N$. This definition allows having a certain task more than once in a task network.

Planning starts with a special task network defining the objective of the problem called *initial task network* tn_I.

The decomposition rules are called *(decomposition) methods* M. They map a task $c \in C$ to a task network, i.e. they are pairs (c, tn). When a method (c, tn) is applied to a task t with $\alpha(t) = c$ in a task network, the task is deleted from the network, the tasks defined in tn are added and they inherit the ordering relations that have been present for t. A task network $tn_1 = (T_1, \prec_1, \alpha_1)$ is decomposed into a task network $tn_2 = (T_2, \prec_2, \alpha_2)$ by a method (c, tn), if tn_1 contains a task $t \in T_1$ with $\alpha_1(t) = c$ and there is a task network $tn' = (T', \prec', \alpha')$ equal to tn but using different ids (i.e. $T_1 \cap T' = \emptyset$). tn_2 is defined as follows:

$$tn_2 = ((T_1 \setminus \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \setminus \{t \mapsto c\}) \cup \alpha')$$
$$\prec_D = \{(t_1, t_2) \mid (t_1, t) \in \prec_1, t_2 \in T'\} \cup$$
$$\{(t_1, t_2) \mid (t, t_2) \in \prec_1, t_1 \in T'\} \cup$$
$$\{(t_1, t_2) \mid (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\}$$

We will write $tn_1 \xrightarrow[t,m]{} tn_2$ to denote that tn_1 can be transformed into tn_2 by decomposing a task t contained in tn_1 using the method m. We will write $tn_1 \to^* tn_2$ to denote that a task network tn_1 can be decomposed into a task network tn_2 by using a (possibly empty) sequence of methods.

The elements s_0, g, and δ are defined as before. The objective of the problem is to find an executable decomposition of $tn_I = (T_I, \prec_I, \alpha_I)$ by adhering the decomposition methods resulting in a state satisfying g. More formally, a task network $tn_S = (T_S, \prec_S, \alpha_S)$ is a *solution* if and only if (1) $tn_I \to^* tn_S$, i.e. tn_S can be created by decomposing tn_I, (2) All tasks in tn_S are primitive, and (3) There is a sequence of all tasks satisfying the ordering constraints $\prec_S$, applicable in s_0 that results in a goal state.

An HTN planning system is not allowed to add tasks apart from the decomposition process. Since we defined the HTN problem as an extension of a classical problem, it contains a state-based goal definition as well. Specifying such a goal is optional, and is indeed not required from a theoretical perspective as it can easily be compiled away (Geier and Bercher 2011). Our LM generation procedure works fine without a state-based goal definition and most problem instances in the used benchmark set do not contain one.

We now define various types of landmarks. Our definition for *task landmarks* (Def. 1) is essentially equivalent to Def. 3 of Elkawkagy et al. (2012), but adapted to our formalism. Most notably, the original formalization is based on a lifted formalization (whereas the respective landmarks are still required to be ground). Def. 2 is new.

Definition 1 (Task Landmark)**.** *A task landmark is a task name $n \in N$ such that every sequence of decompositions leading to some solution tn_S contains a task network including the landmark. Thus, each decomposition sequence from tn_I to tn_S has the form $tn_I \to^* tn \to^* tn_S$, where $tn = (T, \prec, \alpha)$ with $t \in T$ and $\alpha(t) = n$.*

Definition 2 (Method Landmark)**.** *A method landmark is a method $m \in M$ such that every decomposition sequence to every solution tn_S contains two task networks $tn_1 = (T_1, \prec_1, \alpha_1)$ and tn_2 such that there is a task $t \in T_1$ and it holds that $tn_I \to^* tn_1 \xrightarrow[t,m]{} tn_2 \to^* tn_S$.*

Our definition of *fact landmarks* is a canonical adaptation of fact landmarks from classical planning (Porteous, Sebastia, and Hoffmann 2001).

Definition 3 (Fact Landmark)**.** *A fact landmark is a fact $f \in F$ such that for every solution tn_S, every linearization executable in s_0 in line with the ordering and resulting in a goal state there is an intermediate state s_i with $f \in s_i$.*

3 Landmark Generation in HTN Planning

The concept of landmarks in HTN-like planning has first been studied by Elkawkagy, Schattenberg, and Biundo (2010). Here, landmarks have not been extracted from states, but from the task hierarchy. They introduced a technique to identify tasks that are contained in all methods $(c, tn) \in M$ decomposing a certain task c by computing the intersection of their subtasks. These tasks are called *mandatory tasks*. However, the empirical evaluation in their paper evaluates the impact of a domain model reduction that is done simultaneously to the mandatory task generation.

[1] C is short for *compound*, a common synonym for *abstract*.

In follow-up work (Elkawkagy et al. 2012) they tested how the computed landmark information can be exploited by introducing and evaluating landmark-based search strategies. These search strategies are tailored to the deployed search algorithm, which prioritizes different methods that belong to the same abstract task similar to SHOP (Nau et al. 2003; Goldman and Kuter 2019). However, whereas the SHOP systems rely on depth-first search and the order of the methods is specified in the model, Elkawkagy et al.'s system uses informed search strategies and computes the methods' order based on the mandatory tasks. The core idea is to prioritize methods with fewer tasks, whereas only *non*-mandatory tasks are considered (as mandatory tasks have to be achieved anyway). So, their work did not yet define a landmark heuristic that can be exploited by standard heuristic HTN planners.

Bercher, Keen, and Biundo (2014) then introduced these ideas to standard heuristic search. They showed how landmarks of a planning task can be computed based on mandatory tasks and used these landmarks for an admissible landmark counting heuristic. To show in which way we extend their landmark heuristic (Bercher, Keen, and Biundo 2014, Def. 1), we reproduce their definition, but simplified and adapted to our notation:

Definition 4 (Mandatory Task-based Landmarks)**.** *Let $\mathcal{P} = (F, A, C, M, s_0, tn_I, g, \delta)$ and $tn_I = (T_I, \prec_I, \alpha_I)$ be an HTN planning problem. For a primitive task $a \in A$, we define the set of mandatory tasks as $MT(a) = \emptyset$ and for an abstract task $c \in C$ it is defined as follows:*

$$MT(c) = \bigcap_{(c,(T,\prec,\alpha))\in M} \bigcup_{t\in T} \alpha(t)$$

A set of MT landmarks LM^{mt} for $\mathcal{P}$ can be computed by:

$LM^{mt} \leftarrow \bigcup_{t\in T_I} \alpha_I(t)$
while LM^{mt} *changes* **do**
  $LM^{mt} \leftarrow LM^{mt} \cup \bigcup_{n\in LM^{mt}} MT(n)$

The generation method collects the tasks contained in all methods that belong to the same abstract task. Thus all tasks that get introduced at deeper levels of abstraction cannot be found unless all methods share some abstract task(s). This, however, could be improved by lookahead techniques as done in early approaches in classical planning.

4 AND/OR Landmarks in HTN Planning

We now introduce HTN landmark generation based on AND/OR graphs, adapting a landmark technique from classical planning to our setting. A main problem when applying techniques from classical planning to HTN planning is the absence of a state-based goal. Since the objective in classical planning is given in terms of such a goal, techniques like landmark extraction usually rely on it. When an HTN problem also includes one, techniques could be directly applied to it, but it is usually not present. One approach to apply classical techniques would be to extract task landmarks for the HTN model and calculate the state-based landmarks of the preconditions of primitive task landmarks.

However, we introduce a more elegant approach that smoothly combines the generation of task, method, and fact LMs in HTN models based on the approach of Keyder, Richter, and Helmert (2010). Their technique extracts LMs from an AND/OR graph representation for delete-relaxed classical planning problems that was introduced by Mirkis and Domshlak (2007). We first introduce the approach of Keyder, Richter, and Helmert (Sec. 4.1), and then show that it can nicely be adapted to HTN planning (Sec. 4.2).

4.1 Extracting Landmarks in Classical Planning Using AND/OR Graphs

We use the definition of AND/OR graphs by Keyder, Richter, and Helmert (2010, p. 2):

Definition 5 (AND/OR Graph)**.** *An AND/OR graph $G = (V_I, V_{and}, V_{or}, E)$ is a directed graph with vertices $V = V_I \cup V_{and} \cup V_{or}$ and edges E, where V_I, V_{and} and V_{or} are disjoint sets called initial nodes, AND nodes, and OR nodes, respectively. A subgraph $J = (V^J, E^J)$ of G is said to justify $V_G \subseteq V$ if and only if the following conditions holds:*

1. $V_G \subseteq V^J$
2. $\forall a \in V^J \cap V_{and} : \forall (v,a) \in E : v \in V^J \wedge (v,a) \in E^J$
3. $\forall o \in V^J \cap V_{or} : \exists (v,o) \in E : v \in V^J \wedge (v,o) \in E^J$
4. *J is acyclic*

Let $P = (F, A, s_0, g, \delta)$ with $\delta = (prec,\ add,\ del)$ be a delete-relaxed (DR) classical planning problem (i.e., for all $a \in A$ holds $del(a) = \emptyset$). It can be understood as the following AND/OR graph (Mirkis and Domshlak 2007; Keyder, Richter, and Helmert 2010):

Definition 6 (AND/OR representation of delete-relaxed classical problems)**.** *Let $G = (V_I,\ V_{and},\ V_{or},\ E)$ with $V_I = s_0$, $V_{and} = A$, and $V_{or} = F \setminus s_0$. The set of edges is defined as $E = \{(a,f) \mid a \in A, f \in add(a)\} \cup \{(f,a) \mid a \in A, f \in prec(a)\}$.*

Landmarks in these graphs are characterized by the following definition (Keyder, Richter, and Helmert 2010):

Definition 7 (Landmarks in AND/OR graphs)**.**

$$LM(v) = \{v\} \text{ for } v \in V_I,$$
$$LM(v) = \{v\} \cup \bigcap_{u\in pred(v)} LM(u) \text{ for } v \in V_{or},$$
$$LM(v) = \{v\} \cup \bigcup_{u\in pred(v)} LM(u) \text{ for } v \in V_{and},$$

where $pred(v)$ is the set of predecessors of v in G, i.e. $pred(v) = \{u \mid (u,v) \in E\}$.

The set of landmarks for a problem is then defined as the set of landmarks for the nodes representing the goal definition g, i.e. $V_G = g$ and we are looking for $\bigcup_{n\in V_G} LM(n)$.

Keyder, Richter, and Helmert calculate the maximal set fulfilling these equations in **P** by initializing the landmark sets of all nodes apart from V_I with all vertices of the graph, i.e. the full landmark set. Nodes in V_I are initialized with its own value. Then the equations given before are used as update rules for the sets until a fixpoint is reached.

4.2 Extracting Landmarks in HTN Planning Using AND/OR Graphs

From a high-level perspective, what is encoded in the AND/OR graph constructed above is that for every state feature that is in the goal condition, there must be (at least) one action that has it as an add effect. When an action is in the graph, its preconditions must also be fulfilled, i.e. there must be at least one action (for each precondition fact) with this state feature as add effect, and so on (until the state features in the initial state are reached).

When we now have a look at HTN planning, we see a similar structure: for each abstract task in the initial task network, there must be a method decomposing it. When a method is in the graph, all its subtasks must also be in the graph, and so on (until all tasks are primitive). This similarity to AND/OR graphs has been pointed out before (Kambhampati, Mali, and Srivastava 1998; Ghallab, Nau, and Traverso 2004, Chapter 11).

However, we do not need to stop at this point: when we have reached an action, we know that its preconditions need to be fulfilled. So there must be actions that have those state features as add effects. To reflect this in landmark generation, we do not replace the definition of the AND/OR graph given before, but extend it in the following way.

Definition 8 (AND/OR representation of delete-relaxed HTN problems)**.** *Let $P = (F, A, C, M, s_0, tn_I, g, \delta)$ be an HTN planning problem. We define the corresponding AND/OR graph as follows:*

$G = (V_I, V_{and}, V_{or}, E)$ with $V_I = s_0$, $V_{and} = A \cup M$[2] and $V_{or} = F \setminus s_0 \cup C$. The set of edges is defined as $E =$

$$\{(a, f) \mid a \in A, f \in add(a)\} \cup$$
$$\{(f, a) \mid a \in A, f \in prec(a)\} \cup$$
$$\{(m, c) \mid m = (c, tn) \in M\} \cup$$
$$\{(n, m) \mid m = (c, (T, \prec, \alpha)) \in M, t \in T, \alpha(t) = n\}$$

Now we generate the landmarks by using the same generation mechanism as Keyder, Richter, and Helmert. Since the size of the graph is linear in the size of the model, it trivially follows that this computation is in **P**. The overall set of LMs is then based on hierarchy and (if present) state-based goal:

Definition 9 (HTN Landmarks)**.** *Let $tn_I = (T_I, \prec_I, \alpha_I)$ be the problem's initial task network. The overall set of HTN and/or landmarks LM^{ao} is defined as*

$$LM^{ao} = \bigcup_{v \in V_G} LM(v) \text{ with } V_G = \bigcup_{t \in T_I} \{\alpha_I(t)\} \cup g$$

The example given in Figure 1 illustrates the interplay of hierarchy and state during landmark generation. The initial task network contains a single abstract task T that may be decomposed using the methods m_1 or m_2, both introducing an action b. The abstract task S can be decomposed into an action a using m_3. There are two state features x and z. The former is included in the initial state ($s_0 = \{x\}$) and precondition of a. The latter is the precondition of b. When we apply the mandatory task landmark generation, we end up with the landmark set $\{T, b\}$.

[2]Wlog., we assume that $A \cap M = \emptyset$ and $F \cap C = \emptyset$.

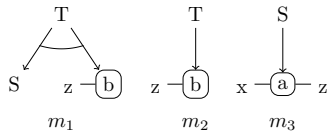
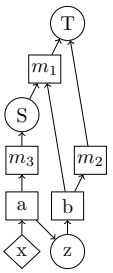


Figure 1: Simple HTN domain. S and T are abstract tasks, m_1 to m_3 methods, a and b actions, and x and z state features. T might be decomposed by m_1 into S and b, or by m_2 into b. S can be decomposed by m_3 into a.



$$\begin{aligned}
LM(x) &= \{x\} \\
LM(z) &= \{a, x, z\} \\
LM(a) &= \{a, x\} \\
LM(b) &= \{a, b, x, z\} \\
LM(m_1) &= \{a, m_3, S, x\} \cup \{a, b, x, z\} \cup \{m_1\} \\
&= \{a, b, m_1, m_3, S, x, z\} \\
LM(m_2) &= \{a, b, m_2, x, z\} \\
LM(m_3) &= \{a, m_3, x\} \\
LM(S) &= \{a, m_3, S, x\} \\
LM(T) &= (\{a, b, m_1, m_3, S, x, z\} \cap \{a, b, m_2, x, z\}) \cup \{T\} \\
&= \{T, a, b, x, z\}
\end{aligned}$$

Figure 2: AND/OR graph of our example given in Fig. 1. Circles are OR nodes, boxes are AND nodes, and the diamond-shaped node labeled x is the only initial node.

The AND/OR graph resulting from the problem is given in Figure 2. The resulting landmark sets are given at the right. Notably, though it is not even reachable when using m_2, we end up with a inside our landmark set, since it is the only action that fulfills the precondition of the landmark b.

4.3 Theoretical Properties

Before we state the properties of *our approach*, let us state theoretical properties of landmarks *in general*. Similar to classical planning, we will see that deciding whether a task, method, or fact is a landmark is as hard as planning itself.

Theorem 1. *Let $\mathcal{P}$ be an HTN planning problem. Let t be a task, m be a method, and f a fact. Deciding whether t, m, or f is a landmark is exactly as hard (with matching upper and lower bounds of the respective complexity class) as deciding the plan existence problem for $\mathcal{P}$.*

Proof. Our proof is a straight-forward adaptation of the corresponding proof by Hoffmann, Porteous, and Sebastia (2004, Thm. 1) for classical planning landmarks.

Hardness. We will introduce a new artificial initial abstract task c_I with two decomposition methods. The first, m_1, decomposes c_I into the original initial task network of $\mathcal{P}$, whereas the other, m_2, decomposes it into a new task network that solves the problem. For this, m_2 decomposes into an abstract task t, which in turn decomposes (i.e., with yet another method) into a primitive task t'. t' uses an empty precondition, a new "dummy" fact f as effect, as well as g as further effects. t' does not use negative effects. Note that we could have put t and t' into the same task network, thus saving another "decomposition level" and method, but we

wanted to keep the size of new task networks limited to 1 so we don not potentially change properties of the problem we reduce from (like number of tasks per task network, their form of ordering constraints, or the "position" of said task, which may all influence the computational complexity).

Clearly, m_2, t (abstract), t' (primitive), and f are landmarks if and only if $\mathcal{P}$ is *un*solvable. Note that determining the unsolvability is as hard as determining the solvability, as those two problems are complimentary to each other.

Membership. Similar to Hoffmann, Porteous, and Sebastia (2004, Thm. 1), we test whether the problem remains solvable if we ignore all parts of the model that "relate" to the landmark(s) in question. I.e., in case of a method we just remove it. In case of a task (abstract or primitive), we remove all task networks and methods that contain them. And in case of a fact landmark, we remove all actions (i.e., again removing all methods that introduce it) that add it. Decide the resulting problem. The task, method, or fact is a landmark if and only if the respective problem is not solvable. □

Using that theorem we can thus deduce the computational hardness of determining whether a task, method, or fact is a landmark for many standard HTN planning problems. We only mention the most important ones here, but as mentioned above our theorem is more general.

Corollary 1. *Let $\mathcal{P}$ be an HTN planning problem. Deciding whether a task, method, or fact is a landmark of $\mathcal{P}$ is **undecidable**. If $\mathcal{P}$ is totally ordered, its complexity is **EXPTIME**-complete. If it is delete-relaxed it is **NP**-complete.*

Proof. Follows from Thm. 1 in conjunction with the results for the general case (Erol, Hendler, and Nau 1996; Geier and Bercher 2011), totally ordered problems (Alford, Bercher, and Aha 2015a), and delete-relaxed problems (Alford et al. 2014; Höller, Bercher, and Behnke 2020). □

We can conclude that any *complete* landmark extraction technique for delete- and ordering-relaxed HTN problems cannot run in polynomial time unless **P**=**NP** (Höller, Bercher, and Behnke 2020).

Some of our further results (i.e., their proofs) rely on the so-called *Decomposition Tree* (Geier and Bercher 2011), which we formally introduce next. It is a formal representation of a task network and it deviation from the initial task.[3]

Definition 10 (Decomposition Tree)**.** *Given an HTN planning problem, a Decomposition Tree (DT) is a tuple $g = (V, E, \prec, \alpha, \beta)$. V and E are the vertices and edges of a directed tree. $\prec$ is a strict partial ordering on V. $\alpha : V \to N$ maps the vertices to (primitive or abstract) tasks from the problem. Vertices that are labeled with abstract tasks are mapped to methods by $\beta : V \to M$.*

A DT is valid *if its root is labeled with the initial task of the problem and for every vertex v labeled with an abstract task c, the following conditions hold:*

1. *It is labeled with a method applicable to c, i.e. $\beta(v) = (c, tn_m)$.*
2. *The task network induced by the children of v in g differs from tn_m only in the task identifiers.*
3. *For all vertices $v' \in V$, the ordering with respect to the children of v is like defined for HTN planning, i.e. for each child v'' the following conditions hold:*
   (a) if $(v, v') \in \prec$ then $(v'', v') \in \prec$
   (b) if $(v', v) \in \prec$ then $(v', v'') \in \prec$
4. *$\prec$ does only contain ordering relations enforced by the conditions 2 and 3.*

Note that there exists a valid decomposition tree for every solution of an HTN problem (Geier and Bercher 2011, Prop. 1), since they simply represent the underlying hierarchy of the respective task network. We can now discuss some properties of the new approach.

Lemma 1. *Let $\mathcal{P}$ be an HTN planning problem, tn a solution task network, and dt its decomposition tree. Then, there exists a justification for the initial task of the AND/OR graph (given in Definition 8) representing dt.*

Proof. Consider the following observations:

1. Task Insertion – Assume we have a justification for an AND/OR graph representation of a classical problem. Assume we want to add additional actions. This results in more AND nodes, but as long as we support their preconditions by other action nodes or the initial state, we get another valid justification.
2. Eliminating Cycles – Geier and Bercher (2011, Sec. 4.1 and 4.2) have shown that – when allowing an HTN planner to insert tasks apart from the decomposition process – cycles in the decomposition structure are not necessary and can be removed. When removed actions have been needed to make the resulting sequence executable, they can be reintroduced via task insertion. While Geier and Bercher use this result to show an upper bound of the size of task networks (for this special class of HTN planning problems), we need it to show the existence of justifications without cycles.

Given a decomposition tree, we know by Obs. 2 that (a) there is a modified tree that (a) contains a subset of the tasks of g, that (b) does not contain cyclic decompositions and that (c) the contained actions can be made executable by task insertion. Now consider the basic structure of a DT: We start by adding all tasks contained in the (acyclic) DT to the justification. Therefore we know that condition 1 for the justifications is fulfilled. For every abstract task, DT it explicitly contains the method used for its decomposition, i.e., we can use this method to add the edges from the abstract task node to the method node, and from the method node to the subtask nodes. For the hierarchical part of the graph, the latter fulfills condition (2) and the former condition (3) of the justification definition. Since our decomposition structure is acyclic, we know that the new graph is.

What is left to show is that there is a justification for the part of the graph representing the state transition system. By Obs. 1 we know we can "add actions" to fulfill the conditions for a justification. Since we started with a valid decomposition tree before we removed cycles, we know that there is a

[3] Problems with an initial task *network* can trivially be compiled into one with just an initial *task* (Geier and Bercher 2011).

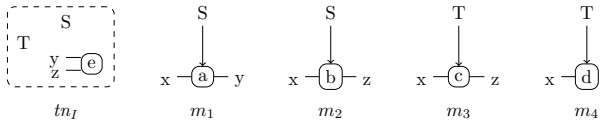


Figure 3: HTN domain without delete-effects and ordering relations. S and T are abstract tasks, a–e are actions, m_1–m_4 are methods, and x–z are state features.

$$
\begin{aligned}
LM(x) &= \{x\} & LM(m_1) &= \{m_1, a, x\}\\
LM(a) &= \{a, x\} & LM(m_2) &= \{m_2, b, x\}\\
LM(b) &= \{b, x\} & LM(m_3) &= \{m_3, c, x\}\\
LM(c) &= \{c, x\} & LM(m_4) &= \{m_4, d, x\}\\
LM(d) &= \{d, x\} & LM(S) &= \{S, x\} = \{S\} \cup\\
LM(e) &= \{a, e, x, y, z\} & & (\{m_1, a, x\} \cap \{m_2, b, x\})\\
LM(y) &= \{y, a, x\} & LM(T) &= \{T, x\} = \{T\} \cup\\
LM(z) &= \{z, x\} = \{z\} \cup & & (\{m_3, c, x\} \cap \{m_4, d, x\})\\
& (\{b, x\} \cap \{c, x\})
\end{aligned}
$$

Figure 4: Landmarks found on the domain given in Figure 3.

set of actions that makes the sequence applicable, thus there is a valid justification for the state transition system. □

Theorem 2 (Soundness)**.** *LM^{ao} landmarks are landmarks for the underlying HTN planning problem.*

Proof. The approach by Keyder, Richter, and Helmert extracts landmarks for AND/OR graphs, i.e., nodes that have to be in *every* justification. By Lemma 1, there is a justification corresponding to every DT. Since every justification includes the nodes, this holds for every one that represents a DT and every DT includes the nodes. □

A second question is whether the approach is complete. Obviously, delete effects and ordering relations are not represented in the graph. Thus all LMs depending on delete-effects and/or the ordering can not be found. This leaves the question whether all LMs apart from these are found for delete- and ordering-free (DOF) HTN problems.

Theorem 3 (Completeness)**.** *LM^{ao} does not find all LMs in DOF HTN planning problems.*

Proof. Consider the HTN domain given in Figure 3. The initial task network (tn_I) contains the abstract tasks S and T, and the action e. The initial state is $s_0 = \{x\}$. All tasks are unordered. S can be decomposed by m_1 into the action a, or by m_2 into the action b. T can be decomposed by m_3 and m_4 into the actions c and d, respectively.

Since e is in tn_I, it is necessarily in every solution. To make it executable, y needs to be fulfilled, thus S needs to be decomposed using m_1 to include a in the plan, which is the only way to make y true. z is also precondition of e, i.e. b or c must be contained in every plan. However, since S needs to be decomposed into a, c is the only option to fulfill z. I.e. c must be contained in the set of LMs.

The LMs found by LM^{ao} are given in Fig. 4. The LMs for the overall problem includes $LM(S) \cup LM(T) \cup LM(e) = \{S, T, e, x, y, z, a\}$. The landmark c is not included. □

The reason for the incompleteness can be seen in the AND/OR encoding. Besides the two obvious relaxations given above (delete-relaxation and ordering-relaxation), a third relaxation is made that further increases the set of solutions: A certain abstract task may be decomposed more than once. This can be seen as task insertion (cf. Geier and Bercher (2011) and Alford, Bercher, and Aha (2015b) for an investigation of its impact on the computational complexity). This relaxation is often used in HTN heuristics to make computation feasible (Alford et al. 2014), e.g. by Bercher et al. (2017) or Höller et al. (2018; 2019; 2020).

The incompleteness result raises the question whether there is a complete algorithm that is feasible (i.e. that can be computed in **P**). However, due to Cor. 1 we know that his is unlikely (as it would require **P**=**NP**). There might, of course, be incomplete methods finding more LMs than ours. However, when we compare our method with the one from the literature, we see that the following theorem holds:

Theorem 4 (Dominance)**.** *Let L_1 and L_2 be the* task *LMs generated by LM^{mt} & LM^{ao}, respectively. Then, $L_1 \subseteq L_2$.*

Proof. In our generation, a task c is represented by an OR node. When its LM set is updated, it is set to the intersection of its predecessors. These predecessors are nodes resulting from the methods m_1 to m_k applicable to c. The LM sets of m_1 to m_k are set to the union of the sets of their subtasks. Since a LM set of a node n contains n by definition, the subtasks of m_1 to m_k contain themselves, i.e. that the sets of m_1 to m_k contain at least all their subtasks, and c the intersection of all these sets. This is exactly the definition of MT LMs. In Fig. 1 and 2 we have given an example for a LM found by LM^{ao} but not by LM^{mt}, so we might find a proper superset of LMs. □

5 Landmarks Apart from HTN Planning

There are many hierarchical planning formalisms in the literature, some of them differ severely in their formalization, semantics, and computational properties (Bercher, Alford, and Höller 2019). Landmarks have also been used in HGN planning, which is concerned with the refinement and achievement of state-based goals rather than tasks (Shivashankar et al. 2012). In a nutshell, there are no *task* networks in HGN planning, but *goal* networks instead, which are partially ordered formulae over state-variables. Methods now refine these goals into further goal networks. There is just one sort of task: the actions known from classical planning. The objective is to find an executable action sequence that achieves all goals and satisfies the given order (possibly by refining goals using the methods), whereas each action can only be applied to a state if it achieves some goal, thereby making it as expressive as HTN planning (Shivashankar et al. 2012; Alford et al. 2016b). When the HGN formalism was first described, Shivashankar et al. (2012) already proposed how an HGN planner could incorporate heuristics, but no landmark information was used.

Their follow-up planner *GoDeL* (Goal Decomposition with Landmarks) (Shivashankar et al. 2013) uses standard classical landmark generation (Hoffmann, Porteous, and Sebastia 2004; Richter and Westphal 2010) to obtain a partially

ordered set of landmarks (called a *landmark graph*) that can be used to guide the search to the next goal.

Their next system *HOpGDP* (Hierarchically-Optimal Goal Decomposition Planner) is guided by a landmark heuristic called h_{HL} (HGN Landmark heuristic) (Shivashankar et al. 2016a; 2016b). h_{HL} builds upon existing landmark techniques and extends them to work on a partially ordered set of goals rather than on a single "at end" goal as in classical planning: Given a current goal network, i.e., a partially ordered set of goals, they can regard this network as a landmark graph and extend it by further landmarks found by techniques from the literature (Richter and Westphal 2010). They then use this extended landmark graph as input to the technique of Karpas and Domshlak (2009) to obtain an admissible heuristic.

The newest HGN system is called *HOGL* (Hierarchically-Optimal Goal Decomposition Planner using LMCut) and again relies on landmarks to guide search (Shivashankar, Alford, and Aha 2017). In this work, they obtain heuristic estimates by first performing a problem relaxation that ignores the hierarchy and action applicability restrictions and then compiling that problem into a classical planning problem (per search node) to use standard classical heuristics. Whereas this approach is agnostic towards the classical heuristic actually used, the approach was evaluated using the LM-cut heuristic (Helmert and Domshlak 2009).

To summarize this line of research: In HGN planning, information about landmarks was used successfully to a large extent, but all approaches use the procedures from classical planning as black box procedures without extending them (at all or) by the information provided by the hierarchy.

6 Evaluation

We evaluate our new LM generation on a widely-used HTN benchmark set. It e.g. has been used by Höller et al. (2018) and Behnke, Höller, and Biundo; Behnke, Höller, and Biundo (2019a; 2019b). It contains 144 problem instances from 8 domains. Experiments ran on Xeon E5-2660 v3 CPUs, 4 GB RAM and 10 min time.

Landmark Extraction. Task LMs are extracted by both generation procedures. Over all instances, our generation finds 13% more task LMs than MT. Besides task LMs, our approach also extracts fact and method LMs. However, we find only very few method LMs (0 to 1 per instance)[4]. When we compare the full sets of LMs that are found (Figure 5), we extract 2.3 as many LMs over the entire instance set.

Extraction time is not an issue for both methods: MT LM generation needs 0.03 ms on average, we need 1.3 ms.

Landmark-guided Search. Though the focus of this paper is on LM *generation*, we want to show that the newly found LMs bear information that helps guiding the search. We therefore integrated the generation mechanisms into the PANDA framework and combined them with the progression search algorithm described by Höller et al. (2020, Alg. 3). We realized the following heuristics:

- **LMC-MT** – Landmark count heuristic using MT LMs. Landmarks are extracted once for the initial task network. During search, reached LMs are tracked and the number of unfulfilled LMs is used as heuristic value.
- **LMC-AND/OR** – Same as before, but using our LM generation (which also includes fact and method LMs).
- **LMC-AND/OR-R** – As before with additional analysis checking whether all unfulfilled LMs are still reachable.

Be aware that a configuration with reachability analysis is not reasonable for MT LM generation. Here, all LMs are reached *by definition*, there is no chance to prevent this. Have a second look at Fig. 1 & 2. After applying m_2, a is not

[4]This is caused by the grounding procedure of PANDA (see Behnke et al. (2020)). Whenever there is only a single method m for a task c, occurrences of c in other methods (or the initial task network) are replaced by the subtasks of m. At most a single method LM is left that is caused by a second compilation step that replaces an initial task *network* by an initial *task*.

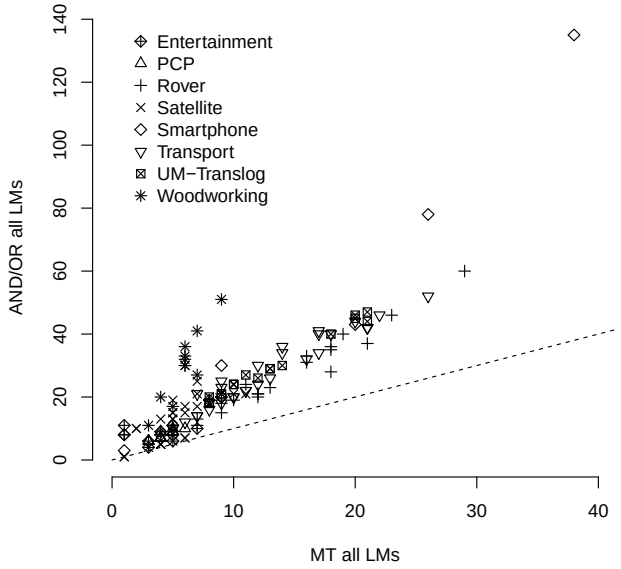


Figure 5: Number of all LMs extracted by MT (on the x axis) and AND/OR (on the y axis) generation split by domain. Please be aware the different scaling of the axis.

	#instances	LMC-AND/OR-R WA*5	LMC-AND/OR WA*5	LMC-MT WA*6	RC FF WA*2	SAT'19	SAT'18	TDG_m WA*2	TDG_c WA*2	2ADL Jasper
ENTERTAINMENT	12	9	9	9	**12**	**12**	**12**	9	9	5
PCP	17	13	13	13	**14**	12	12	9	8	3
SATELLITE	25	21	21	21	**25**	**25**	**25**	24	21	23
SMARTPHONE	7	4	4	4	5	**7**	6	6	5	6
UM-TRANSLOG	22	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**	19
WOODWORKING	11	5	6	6	10	**11**	**11**	9	9	5
ROVER	20	4	4	3	4	**10**	4	5	5	5
TRANSPORT	30	7	1	1	15	**22**	**22**	2	1	19
total	144	85	80	79	107	**121**	114	86	80	85

Table 1: Coverage table for different systems.

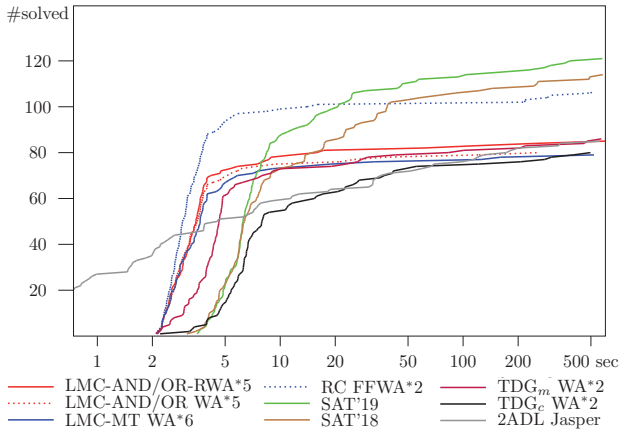


Figure 6: Number of solved instances over time.

reachable anymore and the search node can be pruned. For the MT LM set $\{T, b\}$, however, pruning is not possible.

Table 1 shows the coverage of several HTN planning systems. It contains the configuration with the highest coverage for each of our LM heuristics; the Relaxed Composition heuristic (Höller et al. 2018) with FF (Hoffmann and Nebel 2001) as inner heuristic (RC FF); TDG$_m$ and TDG$_c$ heuristics (Bercher et al. 2017), and compilation-based systems. Two of the latter bound the problem and translate it to propositional logic (see Behnke, Höller, and Biundo (2018; 2019a)). When no solution is found, the bound is increased. The third compilation (Alford et al. 2016a) translates the (also bounded) problem to classical planning and uses the Jasper planner (Xie, Müller, and Holte 2014) to solve it.

It can be seen that the LMC heuristic benefits from the new LM generation process. When only looking at coverage, the possibility to integrate a reachability analysis has a larger impact than the increased LM set. However, as can be seen in Figure 6 (showing solved instances after a given time), the increased LM set also speeds up search considerably. While the SAT-based systems perform best, our new LM generation makes LMC competitive with all search-based systems apart from the RC FF heuristic. However, having the sophisticated and rather complex search techniques of successful LM planners in classical planning like LAMA (Richter and Westphal 2010) in mind, it is not surprising that a simple LM count heuristic is not competitive with the RC heuristic.

7 Conclusion

We introduced a novel LM generation technique for HTN planning that is based on AND/OR graphs. Notably, we do not depend on a state-based goal definition, which is often not present in HTN planning though we can also extract LMs from this definition if there is one. Our approach finds fact, task, and method LMs in a single generation process. It dominates the approach on HTN LMs from the literature, even when restricted to just task LMs (no other kinds of LMs could be extracted before). We have shown that the approach is sound, incomplete, runs in **P**, and that every complete technique must be **NP**-hard. We tested our approach on a widely-used benchmark set and showed that it also finds more LMs in practice. Though the simple LM count heuristic we used is not competitive with state-of-the-art solving techniques, we showed that the new LMs bear information valuable to guide the search.

Acknowledgments

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projektnummer 232722074 – SFB 1102 / Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 232722074 – SFB 1102.

References

Alford, R.; Bercher, P.; and Aha, D. 2015a. Tight bounds for HTN planning. In *Proc. of the 25th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 7–15. AAAI Press.

Alford, R.; Bercher, P.; and Aha, D. W. 2015b. Tight bounds for HTN planning with task insertion. In *Proc. of the 24th Int. Joint Conf. on AI (IJCAI)*, 1502–1508. AAAI Press.

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2014. On the feasibility of planning graph style heuristics for HTN planning. In *Proc. of the 24th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. AAAI Press.

Alford, R.; Behnke, G.; Höller, D.; Bercher, P.; Biundo, S.; and Aha, D. W. 2016a. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 20–28. AAAI Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016b. Hierarchical planning: relating task and goal decomposition with task sharing. In *Proc. of the 25th Int. Joint Conf. on AI (IJCAI)*, 3022–3029. AAAI Press.

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S. 2020. On succinct groundings of HTN planning problems. In *Proc. of the 34th AAAI Conf. on AI (AAAI)*, 9775–9784. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2018. Tracking branches in trees – A propositional encoding for solving partially-ordered HTN planning problems. In *Proc. of the 30th Int. Conf. on Tools with AI (ICTAI)*, 73–80. IEEE Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019a. Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning. In *Proc. of the 33rd AAAI Conf. on AI (AAAI)*, 7520–7529. AAAI Press.

Behnke, G.; Höller, D.; and Biundo, S. 2019b. Finding optimal solutions in HTN planning – A SAT-based approach. In *Proc. of the 28th Int. Joint Conf. on AI (IJCAI)*, 5500–5508. IJCAI Organization.

Bercher, P.; Alford, R.; and Höller, D. 2019. A survey on hierarchical planning – One abstract idea, many concrete realizations. In *Proc. of the 29th Int. Joint Conf. on AI (IJCAI)*, 6267–6275. IJCAI Organization.

Bercher, P.; Behnke, G.; Höller, D.; and Biundo, S. 2017. An admissible HTN planning heuristic. In *Proc. of the 26th Int. Joint Conf. on AI (IJCAI)*, 480–488. IJCAI Organization.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of the 7th Annual Symposium on Combinatorial Search (SoCS)*, 35–43. AAAI Press.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *Proc. of the 26th AAAI Conf. on AI (AAAI)*, 1763–1769. AAAI Press.

Elkawkagy, M.; Schattenberg, B.; and Biundo, S. 2010. Landmarks in hierarchical planning. In *Proc. of the 19th European Conf. on AI (ECAI)*, 229–234. IOS Press.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence* 18(1):69–93.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the 22nd Int. Joint Conf. on AI (IJCAI)*, 1955–1961. AAAI Press.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning – Theory and Practice*. Elsevier.

Goldman, R. P., and Kuter, U. 2019. Hierarchical task network planning in common Lisp: the case of SHOP3. In *Proc. of the 12th European Lisp Symposium (ELS)*, 73–80. ACM.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. of the 19th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 162–169. AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR* 22:215–278.

Höller, D.; Bercher, P.; and Behnke, G. 2020. Delete- and ordering-relaxation heuristics for HTN planning. In *Proc. of the 29th Int. Joint Conf. on AI (IJCAI)*, 4076–4083. IJCAI Organization.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *Proc. of the 21st European Conf. on AI (ECAI)*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 158–165. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. A generic method to guide HTN progression search with classical heuristics. In *Proc. of the 28th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 114–122. AAAI Press.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2019. On guiding search in HTN planning with classical planning heuristics. In *Proc. of the 28th Int. Joint Conf. on AI (IJCAI)*, 6171–6175. IJCAI Organization.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020. HTN planning as heuristic progression search. *JAIR* 67:835–880.

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *Proc. of the 15th National Conf. on AI (AAAI)*, 882–888. AAAI Press.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *Proc. of the 21st Int. Joint Conf. on AI (IJCAI)*, 1728–1733. AAAI Press.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for And/Or graphs. In *Proc. of the 19th European Conf. on AI (ECAI)*, 335–340. IOS Press.

Mirkis, V., and Domshlak, C. 2007. Cost-sharing approximations for h+. In *Proc. of the 17th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 240–247. AAAI Press.

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

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *Proc. of the 6th European Conf. on Planning (ECP)*, 174–182. AAAI Press.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. of the 23rd AAAI Conf. on AI (AAAI)*, 975–982. AAAI Press.

Shivashankar, V.; Alford, R.; and Aha, D. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *Proc. of the 31st AAAI Conf. on AI (AAAI)*, 3658–3664. AAAI Press.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. of the 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 981–988. IFAAMAS.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In *Proc. of the 23rd Int. Joint Conf. on AI (IJCAI)*, 2380–2386. AAAI Press.

Shivashankar, V.; Alford, R.; Roberts, M.; and Aha, D. W. 2016a. Cost-optimal algorithms for hierarchical goal network planning: A preliminary report. In *Proc. of the 8th Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 102–111.

Shivashankar, V.; Alford, R.; Roberts, M.; and Aha, D. W. 2016b. Cost-optimal algorithms for planning with procedural control knowledge. In *Proc. of the 22nd European Conf. on AI (ECAI)*, 1702–1703. IOS Press.

Xie, F.; Müller, M.; and Holte, R. 2014. Jasper: The art of exploration in greedy best first search. In *Proc. of the 8th Int. Planning Competition (IPC)*, 39–42.

Zhu, L., and Givan, R. 2003. Landmark extraction via planning graph propagation. In *Doctoral Consortium of the Int. Conf. on Automated Planning and Scheduling (ICAPS DC)*, 156–160.

Proceedings of the 3^{rd} ICAPS Workshop on Hierarchical Planning (HPlan 2020)

Planning Using Combinatory Categorial Grammars

Christopher Geib
SIFT
319 1st Ave. North, Suite 400
Minneapolis, MN 55401, USA
cgeib@sift.net

Janith Weerasinghe
Tandon School of Engineering
Department of Computer Science and Engineering
New York University
janith@nyu.edu

Abstract

This paper presents a new model of planning based on representing domain knowledge using Combinatorial Categorial Grammars taken from natural language processing. This enables the capturing of plans with context-free expressiveness. It uses the same representation that has previously been used for plan recognition and has been shown to be learnable. Thus it represents a solid link between planning, plan recognition, and natural language processing. The paper also compares our open source implementation to two other well known hierarchical planners.

Introduction and Motivation

This paper is motivated by two issues in AI research. First, the idea that actions in AI planning are functions from states to states is foundational to AI (Fikes and Nilsson 1971). Planning based on decomposition is almost as old (Tate 1977), and has been very successful in deployed systems. However, *methods*, used to define decompositions in hierarchical planning, are not defined as functions from states to states (Ghallab, Nau, and Traverso 2004; Dvorak et al. 2014; Bercher, Keen, and Biundo 2014; Shivashankar, Alford, and Aha 2017).

Second, while the close relationship between reasoning about action and natural language processing (NLP) is well known (Carberry 1990), the representations used for them have remained distinct making their integration ad hoc and require distinct learning algorithms. To address these two issues, this paper provides a functional formalization of planning in terms of Combinatory Categorial Grammars (CCG) (Steedman 2000), a formalism taken from NLP, and a planning algorithm that contributes:

- Formulation of hierarchical planning using function application and composition (rather than decomposition).
- A planner with context-free expressiveness (Aho and Ullman 1992) based on an NLP grammar formalism.
- Task level partial order semantics in planning based on that used in NLP rather than current task level interleaving common in AI planning.
- A total order implementation that plans directly with the lifted first order logical representation.
- Runtimes comparable to other hierarchical planners.

However, perhaps more important than the technical contributions of the planner is the linkage that it represents between planning, plan recognition, and NLP. While the relationship between formal grammars and hierarchical planning is well known, it has previously only been used to prove complexity and expressiveness results for planning (Erol, Hendler, and Nau 1994; Geib 2004; Höller et al. 2014; Behnke, Höller, and Biundo-Stephan 2015; Höller et al. 2016). In contrast, it is central to the contribution of this work that the proposed planning domain representation is a grammar that is exactly the same as that used in prior work on probabilistic *plan recognition* (Geib 2009; Geib and Goldman 2011). While the results of this prior work won't be covered here, using the same problem domain description for both planning and plan recognition (P&PR) represents a significant step in unifying these research areas.

In addition, CCGs have been used for both NLP parsing (Collins 1997; Clark and Curran 2004) and generation (White and Rajkumar 2008). Further, work on learning NLP CCGs has been very successful (Kwiatkowski et al. 2012). Perhaps, most importantly, (Geib and Kantharaju 2018) has adapted the NLP CCG grammar induction algorithms to show initial results in learning plan CCGs of the kind used in this paper. Thus, demonstrating that CCGs can be used for planning is a significant step linking P&PR with NLP parsing and generation using a learnable representation. No other representation has yet demonstrated this.

(Geib 2016), has sketched ideas similar to those presented here. This paper moves beyond that discussion in presenting a formulation of CCGs more tightly connected to prior work. Further, it presents an improved planning algorithm and discusses critical implementation details. Finally, it presents the first data showing state-of-the-art runtimes, and presents a more extensive discussion of the relation to other work.

Background Definitions

To bridge terminological differences between research in NLP and P&PR, these definitions differ slightly from those presented in CCG work on NLP and even those in previous plan recognition work (Geib 2009; Geib and Goldman 2011).

Definition 1.1 *A **state**, $s(\vec{x})$, is a first order logical formula using only conjunction and negation over a set of domain predicates, $\mathcal{P}$, where $\vec{x}$ denotes a possibly empty sequence of unbound variables used in s.*

We denote individual states with individual lower case italic letters (possibly subscripted) (e.g. $a, s_1(\vec{x})$ etc.). When necessary, we will also use lower case italic names for predicates in the planning domain (e.g. *in-hand*($\vec{x}$)) or enumerate the variables as needed (e.g. *in-hand*(x_1, x_2)). We denote the set of all states over the domain predicates $\mathcal{P}$ as $\mathcal{S}_\mathcal{P}$.

We assume agents have invokable *motor programs* that drive their effectors and may change the state of the world.

Definition 1.2 *We define a **motor program**, $\mathbf{mp}(\vec{x})$, as a parameterized function $\mathcal{S}_\mathcal{P} \rightarrow \mathcal{S}_\mathcal{P}$ that models the results of an agent executing one of its parameterized control programs for every state of the domain.*

We will use a vertical bar to denote applying a motor program to a state, resulting in another state, (ie. $\mathbf{mp}(\vec{x})|s_0 = s_1$).

Motor programs differ from well known planning *operators* (Fikes and Nilsson 1971), in that motor program execution is defined for every state of the domain. Operators are usually defined as a limited set of precondition and effect rules. In our implementation, motor programs are also implemented as precondition and effect rules. However, each motor program has an exclusive and exhaustive set of them. This has the effect of making traditional forward or backward chaining much more computationally expensive. It also means that motor programs are always applicable and may encode significantly more information than operators. As such they may encode knowledge about and be able to make predictions about the outcome of their execution in states outside achieving any anticipated goals.

For example a traditional operator for grasping might have preconditions that would prevent its use if the objects were very hot. A motor program would encode the outcome of possibly burning oneself if such an object is grasped. This is knowledge that might be necessary in an emergency but is not relevant for most problem solving domains. Requiring an encoding for every state helps to more fully model our actual causal knowledge of the domain and prevent the intentional or unintentional encoding of biases about the uses of an agent's lowest level control programs.

Thus, motor programs capture all the causal knowledge the system has about the domain. We will use a CCG to encode information about how to build plans to achieve objectives, but first we need to define *planning domains*, *planning problems*, and *solutions*.

Definition 1.3 *A **planning domain**, $\mathcal{D}$, is a four-tuple $\langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$ where:*

- *$\mathcal{O}$ is a finite set of objects in the domain,*
- *$\mathcal{P}$ is a finite set of first order domain predicates,*
- *$\mathcal{S}_\mathcal{P}$, is a finite set of states defined by $\mathcal{P}$ and $\mathcal{O}$, and*
- *$\mathcal{M}$ is a finite set of motor programs defined on $\mathcal{S}_\mathcal{P}$*

Definition 1.4 *A **planning problem** is a triple $\langle \mathcal{D}, s_0, s_G \rangle$ where:*

- *$\mathcal{D}$ is planning domain,*
- *s_0 is an initial state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$, and*
- *s_G is a goal state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$.*

We will use $\sigma_{\vec{x}}$ to denote a set of bindings of domain objects to the variables, $\vec{x}$, and their application to a state or motor program (e.g. $s(\sigma_{\vec{x}})$ or $\mathbf{mp}(\sigma_{\vec{x}})$) to denote their application producing a *ground instance*.

Definition 1.5 *Given a domain, $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$, and problem, $P = \langle \mathcal{D}, s_0, s_G \rangle$, defined on $\mathcal{D}$, a **solution** or **plan** for P is a sequence of motor program instances from $\mathcal{M}$: $[(\mathbf{mp}_1(\sigma_{\vec{x_1}})), ...\mathbf{mp}_n(\sigma_{\vec{x_n}}))]$, such that:*

$$\mathbf{mp}_n(\sigma_{\vec{x_n}})) \; |...| \; \mathbf{mp}_1(\sigma_{\vec{x_1}}))|s_0 = s_G.$$

Thus, a plan is just a sequence of ground motor programs that when executed in the initial state results in the goal state.

Representing Planning Knowledge in CCGs

We are interested in capturing complex, structured knowledge about the possibly multiple effects the motor program may be used to accomplish and the many and varied roles that it can play in plans. This is very similar to the kind of information that needs to be represented about each word in a CCG. We will follow this NLP research in using the term *category* for these knowledge structures as it ties to to the early formal work on category and combinator theory from from mathematical logic(Curry 1977) and provides a foundation for these algorithms. We will denote categories in capitals and their parameters will be treated, like those in states or motor programs, in a parenthesized list or vector (e.g. A, B($\vec{x}$), H-FULL(x_1, x_2)). We will define categories recursively based on two kinds of categories: atomic and complex.

Definition 1.6 *We define an **atomic category**, $\mathrm{A}(\vec{x})$, as a parameterized function from every state in the domain to a state unique to the category. $\mathrm{A}(\vec{x}) : \mathcal{S}_\mathcal{P} \rightarrow \{s_\mathrm{A}(\vec{x})\}$.*

An atomic category defines a constant function achieving a particular state. Thus executing a motor program described by such a category always achieves the defined state. For example, consider the following specification of the category H-FULL with a single parameter for a simple object-moving domain that we will use for our examples. The first line defines its associated state, and the second assigns the motor program **grasp** to it indicating that the motor program can be used to achieve it.

H-FULL(x_1) := [*in-hand*(x_1) $\wedge$!*on-table*(x_1)].
grasp(x_1) $\rightarrow$ [H-FULL(x_1)].

Thus, **grasp**(cup2), would be a function that always results in states where *in-hand*(cup2) is true and *on-table*(cup2) is not. Where informality is possible, we may use atomic categories as identifiers for the states they achieve.

Following the use of categories in natural language CCGs (Steedman 2000), we will define complex categories using two category construction operators, "/" and "\".

Definition 1.7 *Given a set of categories $\mathcal{C}$, where $Z \in \mathcal{C}$ and $\{W, X, Y, ...\} \neq \emptyset$ and $\{W, X, Y, ...\} \subset \mathcal{C}$, we define $Z/\{W, X, Y, ...\}$ and $Z\backslash\{W, X, Y...\}$ as **complex categories**. The category on the left of the slash is called the category **result** and the categories on the right are the **arguments**.*

The slash operators define new categories that combine a set of argument categories to produce a result category. They also define the direction in which the category looks for its arguments, either before or after as determined by the slash. For example, consider extending our example:

H-FULL(x_1) := [*in-hand*(x_1) $\wedge$!*on-table*(x_1)].
H-ARND(x_1) := [*hand-around*(x_1)].
grasp(x_1) $\rightarrow$ [H-FULL(x_1) \ { H-ARND(x_1) }].

This specifies the motor program **grasp** is a function that can be used to achieve the state associated with the atomic category H-FULL, but to do this, immediately before its execution, another function must be executed that results in the state associated with the atomic category H-ARND. Likewise, the forward slash operator requires its argument categories occur after it to produce the state associated with its result category.

Note that the definition of complex categories does not require the use of atomic categories for arguments or results. Thus, complex categories can be built recursively. For example, consider extending our domain fragment for grasping:

release $\rightarrow$ [H-EMTY],
reach4gr(x_1) $\rightarrow$ [H-ARND(x_1)],
grasp(x_1) $\rightarrow$
[((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}],
unreach $\rightarrow$ [H-AT-S].

Using categories, and following NLP terminology, we next define a *Plan Lexicon*.

Definition 1.8 *Given a domain, $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$, we define a **plan lexicon**, $\mathcal{L}$, as a triple $\langle \mathcal{D}, \mathcal{C}^*, \Lambda \rangle$ where,*

- $\mathcal{C}^* = \mathcal{C}_A \cup \mathcal{C}_C$,
- $\mathcal{C}_A$ *= a finite set of atomic categories for states in* $\mathcal{S}_\mathcal{P}$,
- $\mathcal{C}_C$ *= a finite set of complex categories built up recursively starting from* $\mathcal{C}_A$, *and*
- Λ *is a function that maps each motor program in* $\mathcal{M}$ *to a set of categories in* $\mathcal{C}^*$.

We will also refer to such plan lexicons as *plan grammars*. A lexicon augments a planning domain by associating with each motor program a set of categories capturing domain-specific knowledge about how it can be used to achieve specific states. To aid our discussion, we define.

Definition 1.9 *A category, R, is the **root result** of a complex category, C_c, if it is the leftmost, atomic result of C_c.*

For example, PICK is the root result of the category:

((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}.

Definition 1.10 *A motor program, $\mathbf{mp}_i$, is a possible **anchor** of a plan for an (atomic) category, C_a, if the lexicon's Λ maps $\mathbf{mp}_i$ to at least one category whose root result is C_a.*

In our example, **grasp** is an anchor for PICK. Further, we define a *lexical planning problem*.

Definition 1.11 *Given a domain, $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$, and a lexicon, $\mathcal{L}$, defined over $\mathcal{D}$ we define a **lexical planning problem** as a triple $\langle \mathcal{L}, s_0, s_G \rangle$*

- s_0 *is an initial state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$,*
- s_G *is a goal state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$.*

Note that our definition of a lexical planning problem allows for encoding domain knowledge in the lexicon using categories and the Λ function, but solutions are not defined in terms of this knowledge (see Definition 1.5). This distinguishes this work from most prior work on hierarchical planning. We will discuss this in detail later. This said, the new algorithm described in the next section, does make use of such lexically encoded knowledge.

Planning Using CCGs

In addition to viewing a category as declarative knowledge of a motor program's functional role in achieving a goal, we can use its structure to guide the generation of a plan to achieve its root result. Thus, CCG-based planning can be viewed as a recursive structure-following algorithm similar to those used in NLP sentence generation (White and Rajkumar 2008).

Given an atomic category that is a function to a state we wish to achieve, choose a motor program that is one of the category's anchors and add it to the plan, binding any parameters associated with the category. Then recursively build plans to achieve its argument categories, in order, appending the resulting sub-plans either to the left or right of the existing plan as determined by the category's slashes. Note, this builds plans from the anchor motor program outward.

Figure 1 gives pseudocode for this process in a procedure called LEX$_{gen}$ that takes a lexical planning problem as input and returns a plan. To make the category-directed search as clear as possible, we use nondeterministic CHOOSE operators to avoid explicit code for backtracking over category and action choice and parameter bindings. X is a variable over category structures and α is a possibly empty set of categories.

Note that each iteration of the while loop builds a plan for one of the arguments to the category. When all of the arguments in a given set have been processed (see lines 10 and 14) the associated slash is removed allowing the algorithm to access the next set of argument categories (or the root result). Note the resulting plan is tested to verify its success (line 16) before being returned; if not, the algorithm backtracks. Next we will discuss a short example including plan verification.

Figure 2 shows a traditional hierarchical plan structure for much of the same structure captured in CCG 1 . Consider using the lexicon fragment CCG 1 to build a plan to achieve *in-hand*(cup2). First the algorithm must find an atomic category that includes the desired state. PICK satisfies this requirement. PICK's result state and the goal state are unified to find bindings for the category's parameters resulting in PICK(cup2). Figure 3 shows how this instantiated category directs the rest of the plan search. Given a category, the system selects one of its anchors and uses it to bind the motor program's parameters. Line 2 of Figure 3 shows the selection of motor program **grasp** as the anchor for PICK. Parameter binding produces **grasp**(cup2) and a ground instance of **grasp**'s category to direct the rest of the plan search:

((PICK(cup2)/{H-AT-S})\{H-EMTY})\{H-ARND(cup2)}

The system adds the bound motor program to the plan, and then looks at the next argument of the current category, in this

1 Procedure LEX$_{gen}(\mathcal{L}, s_0, G)$ {
2 $\mathcal{C}_G \leftarrow \{ c \in \mathcal{C}^*$ such that $G = root(c)\}$;
3 CHOOSE $c_i(\vec{x}) \in \mathcal{C}_G$ such that $c_i(\vec{x}) \in \Lambda(\mathbf{mp}_j(\vec{x}))$;
4 CHOOSE $\sigma_{\vec{x}}$;
5 $Plan \leftarrow [\mathbf{mp}_j(\sigma_{\vec{x}})]$; $C \leftarrow c_i$;
6 WHILE $(C(\sigma_{\vec{x}}) \neq G)$ {
7 IF $(C = X\backslash Y)$ {
8 CHOOSE c_k such that $Y = \alpha \cup \{c_k\}$
9 $Plan \leftarrow$ APPEND(BuildPlan$(c_k, \mathcal{L})$, $Plan)$;
10 IF $(\alpha = \emptyset)\{C \leftarrow X;\}$ ELSE $\{C \leftarrow X\backslash\alpha;\}\}$
11 ELSE-IF ($C = X/Y$) {
12 CHOOSE c_k such that $Y = \alpha \cup \{c_k\}$
13 $Plan \leftarrow$ APPEND$(Plan,$ BuildPlan$(c_k, \mathcal{L}))$;
14 IF $(\alpha = \emptyset)\{C \leftarrow X;\}$ ELSE $\{C \leftarrow X/\alpha;\}\}$
15 }
16 IF $(Plan|s_0 = G)\{$ RETURN $Plan;\}\}$

Figure 1: Nondeterministic plan generation pseudocode.

case H-ARND(cup2) (see line 3 of Figure 3) and repeats the process. In line 4, the system has selected an motor program that is an anchor for H-ARND and bound it, resulting in:

P = [**reach4gr**(cup2)]

Since H-ARND only requires this single motor program, planning for it is complete. In line 5, because H-ARND was a leftward argument of the category directing the search, the motor program is added to the front of the plan, resulting in:

P = [**reach4gr**(cup2), **grasp**(cup2)]

In lines 6 to 8, the same process is repeated on the H-EMTY category, giving the plan fragment:

P = [**release**, **reach4gr**(cup2), **grasp**(cup2)]

Lines 9 to 11 in the figure repeat the same process for H-AT-S, with the difference that a rightward-looking argument to the category directs the search. As a result, on line 11 the **unreach** motor program is added to the end of the current plan:

P = [**release**, **reach4gr**(cup2), **grasp**(cup2), **unreach**]

Since **grasp**'s category has no more arguments, the plan is

CCG: 1

PICK(x_1) := [*in-hand*(x_1)],
H-FULL(x_1) := [*in-hand*(x_1) $\wedge$!*on-table*(x_1)],
H-EMTY := [!*in-hand*(x_1)],
...
release $\rightarrow$
[H-EMTY, (PLACE(x_1)/{H-AT-S})\{H-ABV(x_1)}],
reach4gr(x_1) $\rightarrow$ [H-ARND(x_1)],
reach4pl(x_1) $\rightarrow$ [H-ABV(x_1)],
unreach $\rightarrow$ [H-AT-S],
grasp(x_1) $\rightarrow$
[((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}],
orient(x_1) $\rightarrow$ [FACE(x_1)],
move(x_1, x_2) $\rightarrow$
[(((MOVE-OBJ(x_1, x_2)/{PLACE(x_2)})/{FACE(x_1)})...
\{PICK(x_2)})\{FACE(x_1)}].

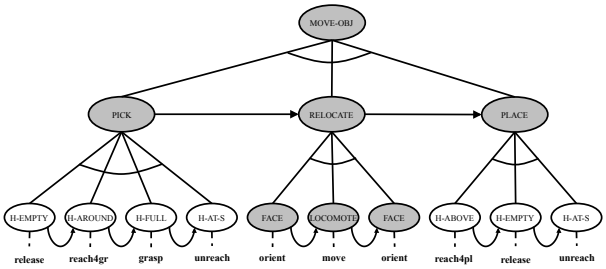


Figure 2: The hierarchical plan to move an object given in CCG 1. Shaded nodes indicate the steps covered by the **move** motor program's complex category.

done. Again note, each argument's sub-plan is added either to the beginning or end of the plan as dictated by the category, building the plan recursively from the middle outward.

Verifying the Plan Categories here are intended to encode functional knowledge that is *likely*, not guaranteed, to hold in all world states. Motor programs are defined in all states of the world, and are executable even where they do not achieve the states specified in their categories.

Consider the first example CCG that mapped **grasp** to H-FULL. While this is a likely outcome of grasping, the **grasp** motor program may predict that H-FULL will not result for objects that are slippery. As we have said, the causal knowledge encoded in the motor programs can be more complete than the knowledge of how to go about building plans to achieve goals encoded in the categories. As such, LEX$_{gen}$'s algorithm is not guaranteed to produce a successful plan by construction and the plan must be verified using the motor programs. If this test fails, the algorithm backtracks across chosen categories and motor programs. Thus, it uses the **mp**'s verify if the constructed plans achieve the goal.

Some might argue against building possible plans and testing them for validity afterwards. However, these objections rest on an empirical question; is this approach actually less efficient than building plans that are valid by construction? The answer to this question is dependent on the speed of the algorithm and the CCG-encoded domain knowledge.

Formal Properties of LEX$_{gen}$

Theorem 1.1 *(Soundness) Let PP= $\langle \mathcal{L}, s_0, s_G \rangle$ be a planning problem. If a trace of LEX$_{gen}$ returns a solution π, then π is a solution to PP.*

Proof Sketch: The planner algorithm works by generating a sequence of motor program instances using a category with the goal as its root result that is within the yield of the grammar. It tests if it is a solution before returning it. (See the last line of the pseudo code.) As such, any plan returned by the algorithm must be a solution to the problem. □

Although the algorithm is sound, it is not is not complete. Since solutions are not restricted by the information encoded in Λ, the system's completeness is contingent on the completeness of the lexicon relative to the planning problem. That is, if the lexicon encodes all of the possible solutions to the problem within its yield then the algorithm is complete.

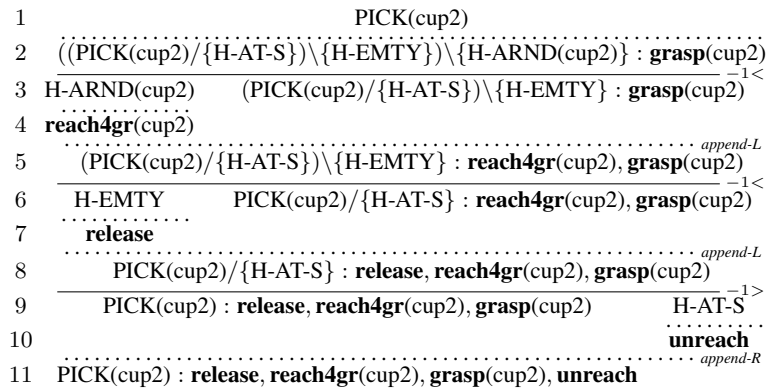


Figure 3: Building a plan to achieve the goal category PICK(cup2). Solid lines denote deconstructing a complex category (-1 and a direction indicator). Dotted lines separate two subtasks of the planning process: choosing a motor program to achieve a particular category (no annotation) and adding the chosen motor program to the plan ("append-L" or "append-R") .

However, we can imagine lexicons that simply don't have a plan within their yield to address a goal. In such cases, the algorithm would fail to produce a plan even if a plan could be assembled from the available motor programs. Thus we can claim only a qualified completeness for the algorithm.

Definition 1.12 *Given a lexical planning problem PP* $= \langle \mathcal{L}, s_0, s_G, \rangle$*,* $\mathcal{L}$ *is defined to be* ***complete with respect to*** *PP if it holds that: if* π *is a solution to PP, then* π *is in the yield of the CCG defined by* $\mathcal{L}$*.*

Theorem 1.2 *(Contingent Completeness) Let PP* $= \langle \mathcal{L}, s_0, s_G \rangle$ *be a planning problem where* $\mathcal{L}$ *is complete with respect to PP. If* π *is a solution for PP, then there is a nondeterministic trace of LEX*$_{gen}$ *that returns* π*.*

Proof Sketch: The yield of any CCG is determined by the categories chosen (and hence the motor programs chosen) and the binding of its arguments to produce instances. Since LEX$_{gen}$ nondeterministically searches over all such possible choice points, there is at least one trace of LEX$_{gen}$ that results in each element of Λ's yield, and since Λ is complete with respect to *PP*, π must be the result one such trace. □

A short discussion of the expressiveness of a CCG plan lexicon is also worthwhile. The expressiveness of CCG grammars is well studied in NLP and is known to depend on the set of *combinators* (Curry 1977) used to combine categories for recognition. Following prior work on plan recognition using CCGs (Geib 2009) our system uses only three combinators:

right application: $X/\alpha \cup \{Y\},\ Y \ \Rightarrow\ X/\alpha,$
left application: $Y,\ X\backslash\alpha \cup \{Y\} \ \Rightarrow\ X\backslash\alpha,$ and
right composition: $X/\alpha \cup \{Y\},\ Y/\beta \ \Rightarrow\ X/\alpha \cup \beta.$

X and Y are categories, and α and β are possibly empty sets of categories. Intuitively they capture the named functional operations respecting the directionality of the category's slash operator. Limiting CCGs to these three operators results in context-free languages (Kuhlmann, Koller, and Satta 2015) making LEX$_{gen}$'s representation less expressive than some other planners like SHOP2 (Nau et al. 2003).

Implementation Details

A short discussion of some details of our first-order, C++ implementation of these ideas is also valuable. First, the CCG-directed search is implemented using iterative deepening allowing it to build recursive plans. Given a recursive category (e.g. $A\backslash\{A\}$) the algorithm searches for plans of increasing length preventing infinite regress or finding suboptimal plans. This represents an improvement over planners that require a fixed bound on search depth or preclude recursive hierarchical plans (Dvorak et al. 2014).

Second, we found it helpful to require LEX$_{gen}$ models be formulated so as to not require a single object be bound to two different parameters in a predicate, category, or motor program. This allows LEX$_{gen}$'s algorithm to remain complete while reducing the binding search space by eliminating from consideration bindings of motor programs such as **drive**(truck1, loc1, loc1), that would drive truck1 from a location to the same location. In cases where this is required, the model and lexicon can be extended with a specialized instance (e.g. **drive-cycle**(truck1, loc1)). Thus, this does not effect the system's expressiveness or completeness and reduced runtimes by a factor of ten. We have found no domains in which this caused an increase in the runtime of the system, however a complete theoretical investigation of this is an area for future work.

Third, the mapping that defines each **mp** is deterministic. One could imagine nondeterministic models that predict a distribution over resulting states. However, this is an area for future work. Finally, our implementation has an implicit **observe** motor program for all atomic categories that are true in the world, this prevents the building of plans to achieve already existing states. Thus given a category like $A\backslash\{B\}$, if B is already true in the current world state, the planner does not attempt to build a plan for it.

Finally, like other hierarchical planners, for goal states that are a conjunction of multiple predicates, we found it helpful to construct specialized complex categories to achieve conjunctive goals. For example, consider a typical goal state in the blocks world (Gupta and Nau 1992) that

looks like *on*(block1, block2), on(block3, block4) and assuming that the category STACKED-ON(x_1, x_2) has the state *on*(x_1, x_2), we could add to the planner the complex category: $G\backslash\{\text{STACKED-ON}(b_1, b_2), \text{STACKED-ON}(b_3, b_4)\}$ to simplify the search for a plan.

Relation to Previous Work

We will focus this discussion on specific technical issues.

Methods vs. Categories: Almost all prior work on hierarchical planning, uses *methods* to define how tasks are decomposed into a series of less abstract subtasks and eventually ground in executable operators. Given a set of task names, $\mathcal{T}$, a method is often defined as a four-tuple $\langle t_h, Pre, T_n, \prec\rangle$ where $t_h \in \mathcal{T}$ is the name of the task this method expands, Pre is a precondition defining when the method is applicable, $T_n \in \mathcal{T}$ is a set of tasks that will replace t_h in the plan, and $\prec$ is a set of ordering constraints on the tasks in T_n for the resulting plan to be a valid. The work on Hierarchical Goal Networks (HGNs) (Shivashankar et al. 2012), also use methods to describe plan knowledge, however the names in $\mathcal{T}$ refer to goal states. CCGs as formalized here are most similar to such HGNs since atomic categories represent functions to states. This said, setting aside method preconditions that will be discussed later, we can think of HTN/HGN methods as encoding a set of context-free grammar (CFG) production rules where the t_h is the left side of the rules and each of the possible orderings of T_n is the right side of a rule.

Given this, and CCG's context free expressiveness, it should not be a surprise that each assignment of a motor program to a CCG category can be rewritten as an equivalent, precondition free HTN/HGN method. The method will have exactly one executable motor program in their right hand side. The root result of the original category defines the thing to be expanded and the argument categories the expansion arrayed around the assigned motor program. For example,

move(x_1, x_2) $\rightarrow$
[(((MOVE-OBJ(x_1, x_2)/{PLACE(x_2)})/{FACE(x_1)})...
$\backslash${PICK(x_2)})$\backslash${FACE(x_1)}].

(with some abuse of notation) could be seen as the precondition free HTN/HGN method, M_1,

$M_1 =$
$\langle t_h = \text{MOVE-OBJ}(x_1, x_2), Pre = [], T_n = \{\text{PICK}(x_2),$
FACE(x_1), **move**(x_1, x_2), FACE(x_1), PLACE(x_2)$\}$,
$\prec = \{(1,2),(2,3),(3,4),(4,5)\}\rangle$.

We note, while this instance is totally ordered, CCGs can represent task level partial order plans.

Since every HTN/HGN method produced by converting CCG categories will contain an executable motor program, CCGs can be seen as an alternative syntax for a generalized version of Greibach Normal Form Grammar (GNFG) (Greibach 1965). In GNFs a single terminal begins the right hand side of every production rule. CCGs are more general in that the terminal can occur anywhere in the rules right hand side. Similar to a GNFG, each motor program category pair can be thought of as storing the results of precompiling search in the decomposition space. Each such pair defines a tree spine from a root to a leaf node. This highlights another difference between CCGs and HTN/HGN methods. A CCG category's can be thought of a slicing the plan tree vertically while HTN/HGN methods slice horizontally. Thus a category's argument categories will most often be at multiple levels of abstraction. This contrasts with HTN/HGN methods that usually capture only one level of plan decomposition and do not require terminals in their expansion.

Further, while a motor program, category pair can be uniquely converted to a HTN/HGN method and thus a CCG plan lexicon could be converted for use by an HTN/HGN planner, a given set of HTN/HGN methods is not uniquely convertible to a CCG plan lexicon. The compiled plan-space search captured in the motor program, category pairs can be done in multiple ways while still producing a complete grammar with the same yield. For example, **release** could be the anchor for MOVE rather than **move**. This would result in a very different category (with only rightward argument categories), that is equivalent to a very different HTN/HGN method. Further it would require a very different rest of the lexicon in order to keep the yield the same. (Geib 2009) has shown that the choice of which motor programs anchor which categories can have a profound effect on the efficiency of plan recognition. We believe a similar effect holds in planning resulting in the earlier pruning of plans that cannot be ground and enabling the early termination of search by producing a complete plan prefix. This is an area for future work.

Preconditions: Many, if not most, other decompositional planners support preconditions restricting a method's application. Such preconditions present theoretical difficulties since they may play at least three different roles: 1) defining causal enablement conditions on the method, 2) preventing a method's use when it is unlikely to result in a successful or desirable plan, and 3) variable binding and search control.

In fact, true causal preconditions (case 1) are very rare. Far more often preconditions are used to control search preventing the application of a method where the domain designer knows it will lead to excessive search or to bind method parameters to reduce search. While LEX$_{gen}$ does not support search control preconditions, true causal preconditions for a CCG are easily encoded in a category by adding leftward looking argument categories.

Total vs. Partial Order: LEX$_{gen}$ is a total order planner similar to (Shivashankar et al. 2012). That said, LEX$_{gen}$'s CCGs do capture task level partial ordering. That is, two argument categories can be specified as partially ordered but their plans cannot be interleaved. First one must be done and then the other. This keeps it in line with CCG use in NLP. Note it does NOT require the unfolding of all orderings of the argument categories in the CCG.

Task Insertion: Unlike several of the latest decompositional planners (Höller et al. 2014; Alford et al. 2016), LEX$_{gen}$does not support task insertion which allows these planners to add actions outside of a known decomposition method. Enabling this in LEX$_{gen}$is an area for future work.

Heuristic Search: (Shivashankar et al. 2012; Shivashankar, Alford, and Aha 2017) provide heuristics for choice points in their decompositional planners. We believe these or similar heuristics can be used to improve LEX$_{gen}$'s search, and will discuss this further in the context of our experimen-

tal results. However, this is still an area for future work.

Status of Domain Knowledge vs. Solutions: It should be clear by now that CCGs in this formulation represent general knowledge about how plans are to be built rather than inviolate knowledge about the causal domain relations. In this, it is more similar to planners that allow for task insertion that see methods as advice for plan construction. This is in sharp contrast to most prior work that defines the correctness of plans in terms of the methods. However, in a deep sense, to define plan correctness this way doesn't actually solve the problem, instead it pushes it into the grammar to which it is more explicitly linked in our formulation. Given a set of motor programs, it is possible that the yield of a grammar does not include plans for all reachable states in the domain. Thus it must be that either we claim that 1) some states are not acceptable goals, 2) the grammar is incomplete or 3) force the grammar to have a yield that reaches every possible state and thereby minimize the aid it might provide.

We feel our formulation makes this issue clearer and is more accurate. We recognize there may be plans an agent has the motor programs to achieve, but hasn't leaned the CCG for its construction. To claim such a planner is complete before this learning is done seems, to us, counter intuitive. Further, this approach preserves the NLP distinction between syntax (CCG categories) and semantics (motor programs) that is critical to the alignment of these area. If a plan is defined as being correct solely because it is in the yield of the grammar it would be equivalent to saying that only syntactically correct sentences could have any meaning which daily experience refutes.

Experiments

We have compared this approach to planning to two other state of the art planners. Note that given the use of CCGs in plan recognition and NLP research, our objective in doing this is not to show that our system always performs better but that it has comparable performance to state of the art planners and therefore is attractive as an integrative framework. We have compared our implementation of LEX$_{gen}$ with the hierarchical SHOP2 planner (Nau et al. 2003) and the ICARUS system (Langley and Choi 2006). We have compared the three systems using twenty two different problems that fall across four domains: blocks world (Gupta and Nau 1992), the satellites domain from the international planning competition (IPC), the logistics domain (Veloso 1992), and a robot-based kitchen domain.

We have chosen not to compared our system to non-hierarchical planners like FF (Hoffmann and Nebel 2001) for several reasons. First, it is generally agreed in the community, that with sufficient domain knowledge hierarchical planners will outperform non-hierarchical planners on large problems. This has a tendency to reduce such comparisons to knowledge and domain engineering competitions. Second, given the universal applicability of motor programs as opposed to operators, FF-style planners might be at a significant disadvantage given the breadth of their search space. Allowing the FF encoding of the domain to reduce the motor-programs back to being operators again opens the question of domain engineering and a level playing field. Third, most FF-style planners uses a propositional representation. In order to be consistent with prior work in NLP, LEX$_{gen}$ works on a first order representation opening questions about counting the cost of grounding. Fourth and finally, while CCGs are well established in the NLP community, we know of no work that suggests the use of non-hierarchical representations for either parsing or generation of language. Thus, even exceptional performance against non-hierarchical planners, while possibly of theoretical interest, would in no way strengthen our argument that this representation represents an important link between work on P&PR and NLP.

Note both SHOP2 and ICARUS use of preconditions to direct method search make them more powerful than LEX$_{gen}$. Further SHOP2 supports the use of "assert" and "retract" operators on its model of the state. This can enable SHOP2 to entirely reformulate a problem before solving even adding new predicates to the domain. In fact, this capability is used to great effect in prior work to produce runtimes that show almost no increase as problems significantly increase in size. However, it raises the question of what limits are placed on this capability. To compare like with like, none of the SHOP2 domains we tested used assert and retract.

We have encoded domains for all three systems with the same level of causal domain knowledge (converting causal preconditions to argument categories in the CCG). We have not used the assert and retract actions in SHOP2, but we have included the results of SHOP2 and ICARUS domains making use of non-causal precondition-directed method decomposition search even though LEX$_{gen}$ does not support such preconditions. For all three systems, the motor programs ("operators" in SHOP2 and "basic skills" in ICARUS) are identical, including the parameter lists, and the same logical propositions describe world states. The same plan structures were encoded in the SHOP2 methods, ICARUS complex skills, and LEX$_{gen}$ complex categories, however as we have discussed the assignment of motor programs to LEX$_{gen}$ categories effectively stores precompiled search in a way not easily replicated in the other systems. We believe this encoding aided LEX$_{gen}$'s performance. A complete study of this is an area for future work.

In Figure 4, "SHOP2–" refers to the execution time of the planner without "assert" and "retract" actions and method preconditions and SHOP2 refers to domains with method preconditions. The execution times reported under ICARUS are for domains with the basic ICARUS skills, the runtimes under ICARUS$'$ contains the same complex skills without preconditions, and runtimes reported under ICARUS$''$ are for domains with complex skills with preconditions.

The satellites domain involve plans for taking one, two and three images. We conducted two types of blocks world tests. The single goal tests involve domains having one to six blocks in the domain with a goal of having a specified block on top of another (the problems were designed such that the target blocks were at the bottom of two stacks). The multiple goal tests included three to five blocks with three to five conjunctive goals. The logistics domain problems included transporting one to three packages within the same city and across two different cities. The four problems in the

Domain	LEX$_{gen}$	SHOP2–	SHOP2	ICARUS	ICARUS′	ICARUS″
Satellites 1 image	0.0342	0.0365	0.0355	4.6037	10.6581	4.0650
Satellites 2 images	0.2408	0.0734	0.0362	6.0789	16.1588	6.3339
Satellites 3 images	53.7502	10.266	0.051	7.6421	25.6157	8.4330
Blocks 3 single goal	0.0003	0.0959	0.0353	6.3337	13.6586	2.8048
Blocks 4 single goal	0.0011	1.4492	0.0352	9.7077	12.1976	3.4093
Blocks 5 single goal	0.0390	—	0.0355	19.0961	36.4703	4.3555
Blocks 6 single goal	0.0717	—	0.0363	26.8817	89.5109	5.3468
Blocks 7 single goal	133.614	—	0.0364	25.7804	89.5109	6.4030
Blocks 3 multi goal	0.0098	—	0.0362	4.0956	49.4053	4.8493
Blocks 4 multi goal	0.0016	—	0.0371	21.5772	11.2691	10.1875
Blocks 5 multi goal	0.0110	—	0.039	37.0607	—	24.3125
Blocks 6 multi goal	0.0645	—	0.0382	26.9061	18.6062	5.3530
Blocks 7 multi goal	143.685	—	0.0385	25.7329	89.4028	6.4028
Logist 1 pack, 1 city	0.0003	0.0345	0.0344	2.2329	2.3463	2.5730
Logist 2 pack, 1 city	0.0017	0.0353	0.0348	4.2632	8.0660	4.2977
Logist 3 pack, 1 city	0.0008	0.0353	0.0353	11.0704	9.6281	9.910
Logist 1 pack, multi city	12.5573	0.0793	0.0371	11.9139	49.2183	48.9495
Logist 2 pack, multi city	568.6783	—	113.037	—	143.136	66.1528
Robot Table Setting 1	0.0019	0.0359	0.036	0.1504	0.1944	0.1748
Robot Table Setting 2	0.1265	0.0386	0.039	0.3932	0.3048	0.326
Robot Table Setting 3	6.4188	0.1701	0.0538	0.232	0.3576	0.3408
Robot Kitchen mixing	0.4657	0.0401	0.0359	—	6.3292	0.5552

Figure 4: Runtimes in seconds for twenty-two problems across four domains for LEX$_{gen}$, SHOP2, and ICARUS.

robotic kitchen domain involve setting the table and mixing ingredients to make a cake. All times are real/wall clock times in seconds. Dashes (—) are runtimes over ten minutes.

Our hypothesis was that LEX$_{gen}$ would perform at about the level of SHOP2 and ICARUS without search control method preconditions, however, it easily surpassed this benchmark. Our results show that given the same domain knowledge, LEX$_{gen}$ has the best performance for ten of the twenty two problems and is beaten only by SHOP2 using search directing preconditions in an additional five domains.

As we have already stated, LEX$_{gen}$ does not support preconditions to control search. However, when there are multiple categories with a desired root result, ordering the search among these categories using a heuristic is an obvious area for future work that we have discussed in the previous section. Conditioning this order on the search will provide a similar capability to that shown in SHOP2 and ICARUS. We anticipate conditioning this search on the state of the world and previous successful uses of the category in building plans. That said, these results clearly show LEX$_{gen}$ has comparable performance to these state of the art planners. That said, with search control knowledge provided by method preconditions, SHOP2's runtime is less affected than LEX$_{gen}$ as problem size increases. While this gives us a good reason to consider encoding heuristic category-selection search, it does not suggest an issue with scaling to larger domains.

Further exploration of where SHOP2 and ICARUS out performed LEX$_{gen}$ has revealed cases attributable specifically to specialized reasoning about the binding of objects to action parameters. For example, we might want to prevent that application of method or skill that resulted in the exploration of plans for the moving of certain specific blocks or blocks that are not of a particular shape. Capturing such restrictions in preconditions on methods is very common in hierarchical planning systems. However, to us this seems more amenable to simple typing of the action and category parameters rather than full preconditions. As a result, we are working on extending LEX$_{gen}$ with typed parameters for predicates, motor programs, and categories.

Finally, we note that ICARUS′ (ICARUS with complex skills but without preconditions) sometimes performs worse than ICARUS with just basic skills. We believe this is because ICARUS was designed to have complex skills with preconditions, and without the preconditions, the complex skill definitions add extra overhead to the search process.

Conclusions

This paper has presented a reformulation of planning and a planning algorithm, LEX$_{gen}$, in terms of CCGs, a state of the art learnable grammar formalism taken from NLP (Geib and Kantharaju 2018). This representation is also exactly the same as that used to perform plan recognition in (Geib 2009; Geib and Goldman 2011). It uses these CCG categories to direct the search for plans, organizing all planning knowledge around executable actions making it a very attractive framework for unifying reasoning about action and language.

References

Aho, A. V., and Ullman, J. D. 1992. *Foundations of Computer Science*. New York, NY: W.H. Freeman Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *IJCAI*, 3022–3029. IJCAI/AAAI Press.

Behnke, G.; Höller, D.; and Biundo-Stephan, S. 2015. On the complexity of HTN plan verification and its implications for plan recognition. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 25–33.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS*. AAAI Press.

Carberry, S. 1990. *Plan Recognition in Natural Language Dialogue*. ACL-MIT Press Series in Natural Language Processing. MIT Press.

Clark, S., and Curran, J. 2004. Parsing the wsj using ccg and log-linear models. In *ACL '04: Proceedings of the 42th Annual Meeting of the Association for Computational Linguistics*, 104–111.

Collins, M. 1997. Three generative, lexicalised models for statistical parsing. In *ACL '97: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*.

Curry, H. 1977. *Foundations of Mathematical Logic*. Dover Publications Inc.

Dvorak, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and acting with temporal and hierarchical decomposition models. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI*, 115–121. IEEE Computer Society.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of AAAI-1994*, 1123–1128.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Geib, C., and Goldman, R. 2011. Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, 958–963.

Geib, C. W., and Kantharaju, P. 2018. Learning combinatory categorial grammars for plan recognition. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 3007–3014. AAAI Press.

Geib, C. 2004. Assessing the complexity of plan recognition. In *Proceedings of AAAI-2004*, 507–512.

Geib, C. W. 2009. Delaying commitment in probabilistic plan recognition using combinatory categorial grammars. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1702–1707.

Geib, C. W. 2016. Lexicalized reasoning about actions. *Advances in Cognitive Systems* Volume 4:187–206.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Greibach, S. A. 1965. A new normal-form theorem for context-free phrase structure grammars. *J. ACM* 12(1):42–52.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2):223–254.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:2001.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS*, 158–165. AAAI Press.

IPC. `http://ipc.icaps-conference.org`.

Kuhlmann, M.; Koller, A.; and Satta, G. 2015. Lexicalization and generative power in CCG. *Computational Linguistics* 41(2):215–247.

Kwiatkowski, T.; Goldwater, S.; Zettlemoyer, L. S.; and Steedman, M. 2012. A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *EACL*, 234–244.

Langley, P., and Choi, D. 2006. Learning recursive control programs from problem solving. *Journal of Machine Learning Research* 7(Mar):493–518.

Nau, D.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 3658–3664. AAAI Press.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, 981–988. IFAAMAS.

Steedman, M. 2000. *The Syntactic Process*. MIT Press.

Tate, A. 1977. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 888–893. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc.

Veloso, M. M. 1992. Learning by analogical reasoning in general problem solving. Technical report, DTIC Document.

White, M., and Rajkumar, R. 2008. A more precise analysis of punctuation for broad-coverage surface realization with ccg. In *Proceedings of the Workshop on Grammar Engineering Across Frameworks*, 17–24. Association for Computational Linguistics.

Proceedings of the 3^{rd} ICAPS Workshop on Hierarchical Planning (HPlan 2020)

Stable Plan Repair for State-Space HTN Planning

Robert P. Goldman and **Ugur Kuter** and **Richard G. Freedman**
SIFT, LLC
319 1st Ave N., Suite 400,
Minneapolis, MN 55401, USA
{rpgoldman, ukuter, rfreedman}@sift.net

Abstract

This paper describes our approach, SHOPFIXER, to plan repair in Hierarchical Task Network (HTN) planning. We developed SHOPFIXER in the SHOP3 HTN planning framework, extending SHOP3's HTN language and theorem-proving capabilities in several ways. Unlike many existing HTN plan repair approaches that depend on chronological backtracking, SHOPFIXER uses backjumping techniques to efficiently, correctly and stably repair the hierarchical plans, where stability means with minimal perturbation to the original plan. We describe our new plan repair method and present experimental results in a number of IPC domains, demonstrating that it generates plans with limited perturbations, and that its plan repair is more computationally efficient than replanning. We compare our results with earlier experimental results from Fox, *et al.* on plan repair and plan stability. Our results confirm theirs, and generalize them. Specifically, we generalize their LPG-repair algorithm to handle plan upsets during execution, and evaluate it in such situations.

1 Introduction

Plan repair has been shown to provide advantages over generating new plans from scratch both in terms of planning runtime and in terms of plan *stability* – the amount of plan content that is retained between the original and repaired plans (Fox et al. 2006). Fox et al. demonstrated that plan repair could provide new plans faster, and with fewer revisions, than replanning *ab initio* in the face of disruptions. They use the term "stability" to refer to the new plan's similarity to the old one, by analogy to the term from control theory. In other prior work, the term "minimal perturbation" has been used synonymously. Plan stability is particularly important for human interaction: users are confused by radical changes to plans introduced in response to trivial upsets.

Previous work on plan stability was limited in that it could only handle plan upsets introduced by modifications to the initial state of the plan (Fox et al. 2006). Our work extends Fox et al.'s approach based on methods from existing plan-repair works such as Wilkins and desJardins (2001), Ayan et al. (2007), Bidot, Schattenberg, and Biundo, and Kuter (2012) in order to handle disturbance that can occur anywhere in the course of plan execution. One way in which SHOPFIXER is less general than Fox et al.'s is that it does not handle goal modification. We chose not to do this because we did not have a good metric for the size of a goal modification, and it is even less well supported by PDDL than are disturbances. Also, most other repair work is limited to disturbance handling.

In this paper, we describe SHOPFIXER, a new method for repairing plans generated by the forward-searching HTN planner, SHOP3. Our method uses a graph of causal links and task decompositions to identify a minimal subset of the plan that must be fixed in plan repair. We also extend the notion of plan repair *stability* introduced by Fox et al. (2006), and further develop their methods and experiments, which demonstrated the advantages of plan repair over replanning.

Contributions of this work are as follows:

- We describe a new technique, SHOPFIXER, and our extensions to the well-known totally-ordered HTN planning formalisms (Ghallab, Nau, and Traverso 2004; Goldman and Kuter 2019). This approach uses causal links for plan repair in SHOP3. Unlike previous such methods (Ayan et al. 2007; Kuter 2012) of the same vein, we show that plan repair and stability in SHOPFIXER is sound and complete, and that its flaw detection method is complete, but unsound: a conservative over-approximation.
- We show how to use NCD (Normalized Compression Distance) for evaluating plan stability and demonstrate how this measure refines plan stability significantly over the existing use of *action distance* (Srivastava et al. 2007) in (Fox et al. 2006) (see Figure 5).
- To support experimentation, we have extended Fox et al.'s LPG variant ("LPG-repair") to be able to repair plans upset in the middle of execution: previously it was only able to react to changes in the initial state, conceptually between the completion of planning, and the commencement of execution.
- In three different planning benchmark domains with different characteristics and over 700 planning and plan-repair planning problems in total, our experimental results confirm earlier results on the *general* advantages of plan repair over replanning, both in terms of computational effort, and in terms of plan stability. SHOPFIXER shows more stable and consistent effect on plan stability measured by NCD than does LPG-repair since SHOPFIXER is a lifted planner and can find plan modifications that are not distinguishable to LPG-repair (cf. Figures 6 and 7).

Listing 1: Effects expressions

```
<literal>* |
(forall (<variable>?) <literal>*) |
(when <GD> <literal>*),
```

- Our experimental results support the utility of our replanning method for forward HTN planning. The SHOPFIXER method is somewhat at odds with the method of Höller et al. (2018), which uses a subtly different definition of HTN plan repair. We discuss these differences with the pros and cons of both approaches.

2 Preliminaries

Our work is done in the context of the SHOP3 HTN planner (Goldman and Kuter 2019), successor to the earlier HTN planners, SHOP2 (Nau et al. 2003), and SHOP (Nau et al. 1999). Ghallab, Nau, and Traverso (2004) describe a restricted case of HTN planning called *Total-order Simple Task Network* (TSTN) planning, a formalization of the original SHOP HTN planning algorithm.[1] TSTN is a restricted version of HTN planning in which each method's subtasks are totally ordered, each method's constraints consist solely of preconditions, and no critics are allowed.

TSTN planning is defined over a language, L, the set of all literals in a function-free first-order language over a finite domain of quantification, $\omega(L)$. A *state*, $s \in S(L)$ is an assignment of truth values to every positive literal in L.

A *TSTN Planning domain* for a language, L, is a tuple[2]: $\mathcal{D}(L) = \langle \text{tasks}(\mathcal{D}(L), \text{ops}(\mathcal{D}(L)), \text{meths}(\mathcal{D}(L))\rangle$ of a set of *tasks*, *operators*, and *methods*. Each operator $o \in \text{ops}(domain)$ is a triple

$$o = (\text{name}(o), \text{precond}(o), \text{effects}(o)),$$

where name(o) is a *task* (see below), and precond(o) and effects(o) are sets of literals called o's *preconditions* and *effects*. An *action* α is an instance of an operator. If a state s satisfies precond(α), then α is *executable* in s, producing the state $\gamma(s, \alpha) = (s - \{\text{all negated atoms in effects}(\alpha)\}) \cup \{\text{all non-negated atoms in effects}(\alpha)\}$.

In SHOPFIXER, we extend this definition to PDDL2.1 actions, but without functional and arithmetic expressions. precond(o) is a logical expression as defined as <GD> in PDDL2.1 grammar (Fox and Long 2003). effects(o) is defined as in PDDL 2.1 excluding durative and functional expressions: see Listing 1.

A *task* is a symbolic representation of an activity. Syntactically, it is an expression $\tau = t(x_1, \ldots, x_q)$ where t is a symbol called τ's *name*, and each x_i is an element of $\omega(L)$. If t is also the name of an operator, then τ is *primitive*; otherwise τ is *nonprimitive* (or "complex"): $\text{tasks}(\mathcal{D}) = \text{prims}(\mathcal{D}) \cup \text{comps}(\mathcal{D})$. Intuitively, primitive tasks can be instantiated into actions, and nonprimitive tasks need to be decomposed (see below) into subtasks.

A *method*, m, specifies how to decompose a task:

$$m = \langle \text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)\rangle$$

where name(m) is m's name and argument list, task(m) $\in$ tasks($\mathcal{D}$) is the task m can decompose, precond(m) is a set of preconditions, and subtasks(m) $= (t_1, \ldots, t_j), t_i \in$ tasks($\mathcal{D}$), the *expansion* of task(m), a sequence of subtasks.

Typically, we work with domain *descriptions*, collections of task, operator, and method *schemas*, with variables for some of the name, precondition, and effect arguments.

A *TSTN planning problem* is a three-tuple $P = \langle s_0, T_0, \mathcal{D}\rangle$, where s_0 is an initial state, T_0 is a sequence of ground tasks, the *initial task list*, and $\mathcal{D}$ is a TSTN domain.

If T_0 is empty, then P's only solution is the empty plan $\pi = \epsilon$, and π's *derivation* (the sequence of actions and method instances used to produce π) is $\delta = \epsilon$. We say that δ is executable in the current state, yielding no state changes.

If T_0 is nonempty (i.e., $T_0 = \langle t_1, \ldots, t_k\rangle$ for some $k > 0$), and s_0 is the current state, then let $T' = \langle t_2, \ldots, t_k\rangle$. If t_1 is primitive and there is an action α with name(α) $= t_1$, and if α is executable in s_0 producing a state s_1, and if $P' = \langle s_1, T', \mathcal{D}\rangle$ has a solution π with derivation δ, then the plan $\alpha \bullet \pi$ is a solution to P (where $\bullet$ is concatenation) whose derivation is $\alpha \bullet \delta$. In that case, $\alpha \bullet \delta$ is executable in s_0. If t_1 is nonprimitive and there is a method instance m such that task(m) $= t_1$, and if s_0 satisfies precond(m), and if $P' = \langle s_0, \text{subtasks}(m) \bullet T', \mathcal{D}\rangle$ has a solution π with derivation δ, then π is a solution to P and its derivation is $m \bullet \delta$. The nonprimitive task task(m) is executable, if s_0 satisfies precond(m) and its subtasks(m) are executable in s_0. By induction, the derivation $m \bullet \delta$ is executable in s_0, if δ is executable. A sequence of tasks $t_1 t_2 \ldots t_n$ is executable in s_0 if t_1 is executable in s_0, the state after executing t_1 is s_1, and $t_2 \ldots t_n$ is executable in s_1.

The definition of a derivation, above, defines a *derivation tree* (or, less formally, a *plan tree*), with edges from complex tasks to the subtasks in their expansion, and with the primitive tasks of the plan as their leaves. We extend the plan trees by adding cross edges from primitive tasks that establish literals to the nodes that consume them in preconditions. While the establishers are all primitives, the consumers may be either primitives or complex tasks (note that the complex task as such does not consume the precondition, it is the *method* whose task network is used that has the preconditions).

A *plan disturbance* is an unexpected change in the world state after the execution of a prefix of the plan. A plan disturbance $\xi(\pi)$, for a TSTN plan, $\pi = \langle \alpha_1 \ldots \alpha_n, \rangle$ is a tuple $\xi(\pi) = \langle \text{pred}(\xi(\pi)), \text{effects}(\xi(\pi))\rangle$, where $\text{pred}(\xi(\pi)) = \alpha_k$ is an action in π[3] and effects($\xi(\pi)$) is a set of ground effects as in a STRIPS operator.

We assume an execution model that is a very simple extension of classical planning: if a task's preconditions hold when it is *started*, then the task is executable. An action that is executable transforms the state into a new state by applying its effects to the state in which it was executed. The effects of plan disturbance $\xi(\pi)$ are applied in the state following the execution of $\text{pred}(\xi(\pi))$.

[1] SHOP3 and SHOP2 are more expressive extensions of SHOP.

[2] We leave the language arguments implicit in the future.

[3] We also permit the special value ϵ for disturbances that occur before any actions.

Listing 2: "PDDL methods" for SHOP3.

```
<method> ::= (:method <method-task>
              :method-name <symbol>
              :variables <typed-list (variable)>
              :precondition <GD>
              :task-net <task-net>)

<method-task> ::= (<task-symbol> <task-arg>*)
<task-arg> ::= <name> | [<variable> -<type>]
<task-net> ::= (<task-net-component>+)
<task-net-component> ::= (<task-symbol>
                          <method-task-arg>*)
<method-task-arg> ::= <term>
```

3 Our SHOP3 Extensions

Our approach to plan repair is based on the existing SHOP3 planner (Goldman and Kuter 2019), with some extensions, and previous work on plan repair in UMCP-style HTN planners (Ayan et al. 2007). Two key extensions are the addition of PDDL handling to SHOP3, and the addition of an alternative search engine that can perform backjumping. We add PDDL handling for two reasons: (1) PDDL has a clearer, and easier to handle semantics than does SHOP3's language: in theory SHOP3's language may comply with PDDL semantics, but in practice, it has the full expressive power of a temporal extension of Prolog. (2) Incorporating PDDL allows us to handle benchmarks and make comparisons.

PDDL in SHOP3 Now SHOP3 can plan with both native operators and PDDL actions, and can incorporate PDDL domains in its own domains by reference. Additionally, we have developed "PDDL methods" for SHOP3.

The STRIPS dialect of SHOP3 is as follows: (1) primitives are PDDL typed STRIPS operators (i.e., equivalent to PDDL requirements of `:typing` `:negative-preconditions`), and (2) the method grammar is given in Listing 2. In that grammar, `<GD>` and `<typed list (variable)>` are as defined in the PDDL 2.1 grammar (Fox and Long 2003). `<task-symbol>` is the same as PDDL 2.1's `<action-symbol>`, however additionally each `<task-symbol>` is classified as either *primitive* or *complex*. `<method-task-args>` and arguments in preconditions, if variables, must be elements of the parameter list. `<method-name>` is also equivalent to `<action-symbol>`: it designates a unique method for achieving the `<task-spec>`. This is equivalent to the following limitations:

1. Typed STRIPS dialect of PDDL for the primitives;
2. Conjunctive preconditions for the methods, no SHOP3 special language features.
3. For methods, only variables in the task parameters and in the `:variables` list are scoped over the preconditions and effects (standard SHOP3 has Prolog-style scoping).
4. All tasks will be ground when added to the plan, and all task parameters (including those for complex tasks) will be ground before the preconditions are checked.

We will initially explain our approach for the simple case of the STRIPS dialect, and then will extend to "ADL SHOP3." The ADL dialect of SHOP2 uses as operators PDDL actions in the ADL dialect, and permits ADL-style quantification in method preconditions (but not functional terms or numerical fluents). So we add to the method preconditions grammar the ability to use `forall` and `exists`, and the full set of boolean connectives.

Note that neither of our SHOP dialects supports SHOP3's `:unordered` construct, which permits partial ordering in a method's task network. We are interested in doing so, but this causes issues with the semantics of plan executability that we discuss further in our conclusions.

Backjumping Another key extension to SHOP3 was to add the ability to do backjumping search. Ordinarily, if SHOP3 makes a poor decision at a state, the search process will lead to dead ends, and it must back up to that state and resume searching with a different decision. The simplest approach to "backing up" is *chronological backtracking*: undoing the most recent decision in the search and trying an alternative. However, in some problems, it is possible to determine that the most recent decision was not relevant to reaching the dead end. Backjumping (Gaschnig 1979) exploits such information by skipping over irrelevant decisions and backtracking directly to the most recent *relevant* decision, d, undoing all the decisions above d in the search stack.

4 Plan Repair

The basic idea behind our plan repair approach is very simple: when a disturbance is introduced into the plan, SHOPFIXER will find the minimal subtree of the plan tree that contains the node whose preconditions are clobbered by that disturbance: the *failure node*. If there is no such node, then the disturbance does not interfere with the success of the plan. SHOPFIXER will then repair the plan, starting with the minimal subtree.

To find the minimal subtree around a failure node, SHOPFIXER finds the first task in the plan that is *potentially* "clobbered" (rendered un-executable) by that disturbance, and restarts the planning search from that task's immediate parent in the HTN plan (since that was the point at which that task was chosen for insertion into the plan). This plan repair is done by backjumping into the search stack for SHOP3 and reconstructing the compromised subtree without the later tasks (see the discussion in the conclusions, Section 8). Note that the first clobbered task may be *either* a primitive task or a complex task. Furthermore, if p is the parent of child c in an HTN plan, then p's preconditions are considered chronologically prior to c's, because it is the satisfaction of p's preconditions that enables c to be introduced into the plan: if both p and c fail, and we repair only c, we will still have a failed plan, because after the disturbance, we are not licensed to insert c or its successor nodes.

SHOPFIXER restarts the planning search by backjumping to the corresponding entry in the SHOP3 search stack, which it retains, and updating the world state at that point with the effects of the disturbance. When restarting the planning search, SHOPFIXER "freezes" the prefix of the plan that has

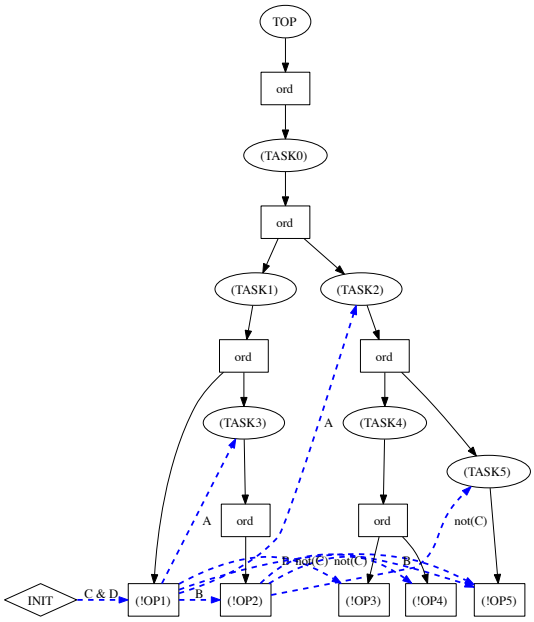


Figure 1: A high-level illustration of causal links over a task hierarchy generated by SHOP3.

already been executed, as well as the deviation and its effects. It may backjump to decisions prior to the deviation, for example, if the immediate parent of the failed task is the top level task of the problem, but it cannot undo the *effects* of an action that is already done. SHOPFIXER returns a repaired plan that is made up of the prefix before the disturbance, the disturbance, and the repaired suffix.

From this simple outline, it can be seen that the critical requirement for plan repair is to find the chronologically first clobbered task. For the STRIPS dialect of SHOP3, we will see that we can provide sound and complete detection of the first clobbered task. However, we cannot do this efficiently for the ADL dialect, because of conditional execution. For the ADL dialect, we can only provide completeness (if a is the first clobbered task in the plan, we will find it or a predecessor), and not soundness (a may appear before the first clobbered task, but no task before it is clobbered). For this reason, it follows from the soundness and completeness of SHOP3 that our plan repair algorithm is sound and complete, but it may do more work than is necessary.

SHOPFIXER finds the first clobbered task using causal links. We have enhanced SHOP3 so that whenever it inserts a task into the plan, it associates with the task causal links to its preconditions. A causal link is a triple, $\langle e, p, t\rangle$, where e is the action that established the ground literal p, and t (the "consumer") is a task that requires p in its preconditions. Figure 1 illustrates an abstract depiction of a graphical representations of causal links over a task hierarchy generated by SHOP3. SHOP3 maintains a link only for the *latest* establisher of p relative to t, as is required for soundness. The causal links are hashed with p as the key, so that if a disturbance clobbers p, SHOPFIXER can find the first consumer.

Given the simplicity of finding the first clobbered consumer from a proposition, p, the soundness and completeness of this task, depends on the soundness and completeness of causal links. Completeness is readily established, since it requires simply recording the establisher of each proposition or, in worst case, searching backwards through the state trajectory of the plan, which SHOP3 maintains (implicitly) for purposes of backtracking/backjumping.

Soundness is also immediate, with the exception of the complex tasks. For a complex task, there may be variables (in the `:variables` property of the method, see Listing 2) not bound until after the method is chosen. Those variables are effectively existentially quantified (their bindings come from a refutation of the negation of the preconditions), and SHOP3 records causal links for the *grounded* preconditions. So, for example, if we have a method like the following:

```
(:method (achieve-goals)
   :variables
     (?obj - objective ?mode - mode)
   :precond
     (communicate_image_data ?obj ?mode)
   :task-net
     (:ordered
       (communicated_image_data
          ?obj ?mode)
       (achieve-goals)))
```

which chooses an objective and a mode from the goals, and SHOP3 chooses `obj1` for `?obj` and `h_res` for `?mode`, then it will record the causal link

$\langle e,$ `(communicated_image_data obj1 h_res)`$, t\rangle$

Assuming that a disturbance countermands this task (i.e., deletes `(communicated_image_data obj1 h_res)`), then arguably treating t as clobbered is incorrect, because the task could be retained, and query for

```
(communicate_image_data ?obj ?mode)
```

rather than replanning its parent. However, in practice, SHOP3 binds these variables concurrently with choosing the task, so if this is unsound, it is a benign unsoundness: when replanning from t's parent, SHOP3 will first consider alternative bindings for `?obj` and `?mode`, and any siblings to the right of this task will be replanned anyway.

We have shown that finding the first clobbered task can be performed in a sound and complete way for the STRIPS dialect of SHOP3. We use this to establish the soundness and partial completeness of the ADL dialect. Recall that ADL adds logical connectives, quantification (`forall` and `exists`), and conditional effects. It is conceptually simple to account for the logical connectives and quantification, because of PDDL's finite domain of quantification:

AND handled as above;

NOT Negated ground literals can be handled identically to positive ground literals, complex negations are rewritten to drive the negations inward;

OR take the causal links from one conjunct or another;

forall over a fixed, finite domain, a conjunction.

exists over a fixed, finite domain, disjunction.

imply this is equivalent to a disjunction.

Note that the use of disjunction already compromises the soundness of finding the first clobberer, because a task with the precondition `(or p q)` that is executed in a state satisfying $p \wedge q$, will get only one causal link, for p or q, so it may be incorrectly retrieved if only one of the two is deleted by a disturbance. In practice, we have not found this unsoundness to be problematic, but it is certainly possible that it would be, particularly in a domain with existentially quantified preconditions ranging over a large set, such as "there must be an employee sitting at a monitoring station," if there are many employees, many monitoring stations, and more than one employee at a single monitoring station.

A second cause of unsoundness comes from conditional effects. Recall that conditional effects are of the form `(when p e)`, where p are the *secondary preconditions* for e relative to the operator with this effect. When SHOP3 adds a to the plan, it will add causal links for p relative to every e that is instantiated, and `(not p)` for every e not instantiated (the latter happens because of the handling of quantifiers that contain the `when` effect). In other words, instead of capturing the causal links for "every effect of a that is used in the plan," SHOPFIXER uses links for "the secondary preconditions that cause a to have *exactly this set* of effects."

We tolerate the unsoundness in order to avoid reasoning about arbitrary logic in preconditions, and chaining conditional effects through multiple stages in a plan. Trying to compute the clobbering conditions exactly would in the worst case involve solving SAT problems. Section 6 illustrates how this tradeoff plays out empirically: to date it seems benign – the savings for plan repair are still substantial, and the cost of maintaining the causal links manageable.

5 LPG Extensions

Recall that Fox, *et al.* (2006) were limited to repairing/replanning only in response to changes in the initial state, rather than changes in the middle of plan execution. We have extended their version of LPG with a preprocessor that handles disturbances in the middle of plans. Using a modified initial state I' and/or goal condition set G' along with plan π that solves an original classical planning problem $\mathcal{P} = \langle F, A, I, G \rangle$, they defined the replanning problem as $\mathcal{P}_{replan} = \langle F, A, I', G' \rangle$ and the repair problem as $\mathcal{P}_{repair} = \langle F, A, I', G', \pi \rangle$. However, just as SHOPFIXER can begin the process at an intermediate state in the plan

$$s_{div} = a_{d-1}\left(a_{d-2}\left(\ldots\left(a_1\left(I\right)\right)\ldots\right)\right)$$

where the divergence occurs just before executing action a_d, we also update the initial state in LPG to be s_{div}. Then the replanning problem is simply $\mathcal{P}_{replan} = \langle F, A, s_{div}, G \rangle$ and the repair problem is $\mathcal{P}_{repair} = \left\langle F, A, s_{div}, G, \pi_{d..|\pi|} \right\rangle$ where $\pi_{i..j}$ is the subsequence of the plan π from actions a_i to a_j, inclusive. That is, LPG begins the planning process in the state where the divergence occurs and repairs the remaining actions in the plan (without the task network information that is available to SHOPFIXER).

6 Experiments

To assess the efficiency and correctness of our approach we have conducted preliminary experiments with the "Openstacks"[4], "Rovers"[5], and "Satellite"[6] domains from the IPC. We chose these domains for their use of the ADL dialect of PDDL. Our results show a substantial savings for using plan repair over replanning from scratch in an unexpected state during plan execution.

Problems for experimentation were constructed from the IPC problems, giving a sliding scale of difficulty for each domain. For each domain, we defined a set of disturbance pseudo-actions, which involved events like shipping failures (in Openstacks), loss of samples, obstructions to line of sight (in Rovers), direction changes, decalibrations, and power loss (in Satellite). The disturbance actions had to be carefully written in order to not render a problem unsolvable, or to change the class of the problem. For the notion of class of the problem, we drew on Hoffmann's (Hoffmann 2005) analysis. For example, we avoided breaking the symmetry of the `can_traverse` relationship in the Rovers problems. This property is not formalized in any way in the domain (PDDL does not support higher-order assertions about problems), but it is present in all of the problems nevertheless. Changing the character of the problems with disturbances would render the repair and replanning less directly comparable. HTN domains for these problems, the problems themselves, disturbance actions, and raw results are available on the web (Goldman, Kuter, and Freedman 2020).

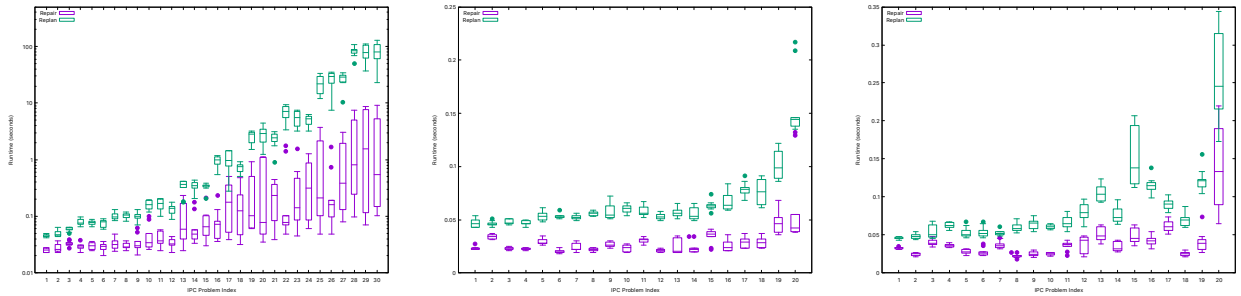
Computational Efficiency Figure 2 shows the comparison between the time to find the original plan and the time to repair the plan for SHOP3. Our experiments with SHOP3 support Fox *et al.*'s argument for plan repair over replanning, and also show the value of our replanning method for SHOP3. For each problem instance along the x-axis, the boxes mark the lower quartile q_1, median, and upper quartile q_3 of the set of times taken per run T. The extended whiskers indicate the interval of times taken per run that are within 1.5 times the interquartile range from these quartiles; that is, a line from the minimum to the maximum of $W = \{t \in T \,|q_1 - 1.5\,(q_3 - q_1) \leq t \leq q_3 + 1.5\,(q_3 - q_1)\}$. Any runs whose times are outside W are marked with a dot. The results here are for 10 runs for each problem instance of Openstacks, Rovers, and Satellite. Each run has a randomly generated deviation that clobbers the preconditions for some action in the plan suffix, rendering the plan un-executable or preventing it from achieving the goal. Note that Openstacks runtimes, unlike those for the other two domains, are plotted on a logarithmic scale, because of their high variance.

Recall that in order to make a plan repairable, SHOP3 must save much more information in the course of planning. In particular, it must build a plan tree that records causal links between actions and the tasks that consume their effects. In order to determine whether our plan repair is simply a tradeoff between spending more time in planning to save time in plan repair, we compare the runtime of generat-

[4] http://icaps-conference.org/ipc2008/deterministic/

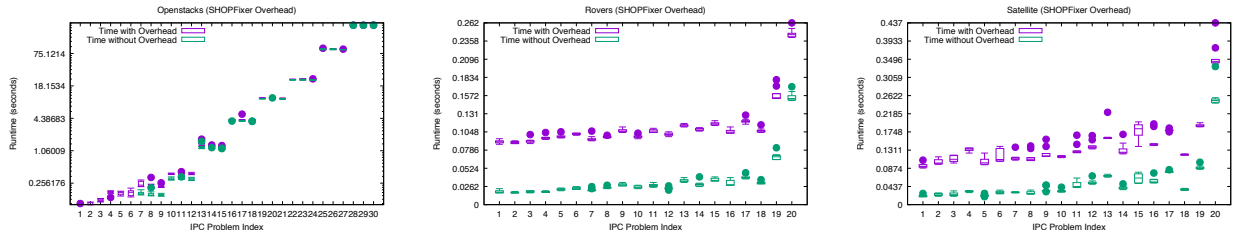
[5] http://ipc02.icaps-conference.org/

[6] http://ipc04.icaps-conference.org/deterministic/



(a) Openstacks problems (b) Rovers problems (c) Satellite problems

Figure 2: Comparing plan repair time to replanning, SHOP3.



(a) Openstacks problems (b) Rovers problems (c) Satellite problems

Figure 3: Overhead when planning for repair-ability, SHOP3.

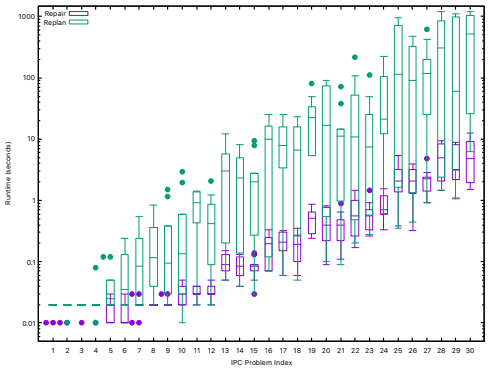


Figure 4: Comparing Openstacks repair time to replanning, LPG.

ing repairable plans with the additional information needed for replanning together with the runtime for generating plans without the additional information. Results are in Figure 3.

We ran the same problems and deviations using LPG-repair (henceforth "LPG"), with the preprocessing described above. Results on Openstacks, in Figure 4, confirm those with SHOP3: they show a high variance (Openstacks runtimes are plotted logarithmically), and show a clear advantage for plan repair, in terms of runtime. Results on Rovers (Table 1) and Satellite on the other hand, are equivocal, showing no clear advantage for repair over replanning. They exhibit a floor effect: runtimes for these problems are not significant for LPG. We have omitted the table for Satellite to save space: it is essentially identical to that for Rovers.

Plan Stability Metric Another proposed advantage of plan repair over replanning is *stability*: repaired plans will be more similar to the original plans than entirely new plans from replanning. In their paper, Fox et al. (2006) measure plan stability using *Action Distance* (AD). Briefly, this is the cardinality of the symmetric set difference of the actions in the two plans. We have argued elsewhere (Goldman and Kuter 2015) that there are a number of problems with this definition: it's insensitive to the ordering of steps in the plan, it treats `(drive truck1 src dest)` and `(drive truck2 src dest)` as just as different as `(drive truck1 src dest)` and `(navigate vaporetto1 src dst)`, *etc.*. We propose the use of *Normalized Compression Distance* (NCD) as a better alternative to AD. For an example of why we prefer NCD, see Figure 5. One can easily see that AD provides a much less stable measure of distance, so we use only NCD going forward.

Plan Stability Our experiments with SHOP3 and LPG confirm and *extend* the Fox et al. (2006)'s results: extend because we support plan upsets anywhere during execution. Our results on plan stability, measured using NCD are shown in Figure 6 for SHOP3, and 7 for LPG. These give distance between repaired and replanned plans and original plans, showing that for both planners repair improves stability.

7 Related Work

Previous extensions of the SHOP framework include HOTRiDe (Ayan et al. 2007) and SHOPLIFTER!(Kuter 2012). SHOPLIFTER augments SHOP2's HTN representations and planning capabilities with a constraint-based formalism for HTNs, inspired by UMCP (Erol, Hendler, and

Problem	Repair Time mean	std	Replan Time mean	std
1	0.02	0.00	0.02	0.00
2	0.02	0.00	0.02	0.00
3	0.02	0.00	0.02	0.00
4	0.02	0.00	0.02	0.00
5	0.02	0.00	0.02	0.00
6	0.02	0.00	0.02	0.00
7	0.02	0.00	0.02	0.00
8	0.02	0.00	0.02	0.00
9	0.02	0.00	0.02	0.00
10	0.02	0.00	0.02	0.00
11	0.02	0.00	0.02	0.00
12	0.02	0.00	0.02	0.00
13	0.02	0.00	0.02	0.00
14	0.02	0.00	0.02	0.00
15	0.02	0.00	0.02	0.00
16	0.02	0.00	0.02	0.00
17	0.02	0.00	0.02	0.00
18	0.03	0.00	0.03	0.00
19	0.03	0.00	0.04	0.01
20	0.05	0.01	0.04	0.01

Table 1: LPG replan and repair times for Rovers domain.

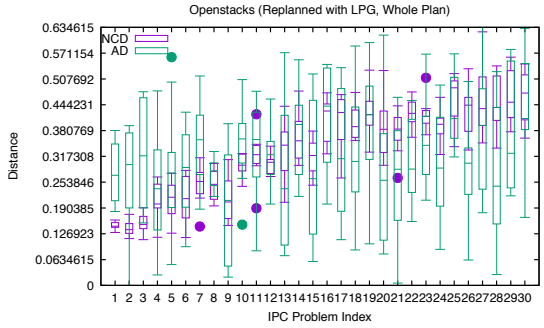


Figure 5: Comparing NCD and AD.

Nau 1994). These constraints provide the required representation conditions that need to hold during the execution of a task network as well as action post-conditions and plan goals. Neither HOTRiDe nor SHOPLIFTER provide the guarantees of correctness we give here.

Warfield et al. (2007) developed a replanning algorithm called RepairSHOP, similar to HOTRiDe. They differ in their dependency representations. RepairSHOP uses a more general and expressive data structure, a "GoalGraph." Although GoalGraphs would enable the planner to produce explanations for task dependencies and replan using those explanations, it is not clear how the two approaches compare in terms of expressive power and efficiency. Unfortunately, RepairSHOP was not available for use in our comparison experiments. Schattenberg (2009) also uses a partial-order causal link (POCL) formalism, a generalization of our totally-ordered causal links. Their repair strategies resemble ours, but do not attempt to maintain stability.

Recent work by Höller, *et al.* (2018) works from a UMCP-like POCL basis, rather than forward planning as we do. They pose the plan repair problem as a constrained HTN planning problem, by transforming the original problem description. This approach allows them to use a "stock" planner for repair, not requiring a separate repair algorithm. Most interestingly, they attempt to fully honor the constraints implied by the task networks, in a way we do not. Arguably this is more correct, but equally it could be argued that this allows only repairs with counterintuitive limitations. Their system requires that all completed actions be part of any task network constructed in repair. We show the difference between our approaches in Figure 8. Consider the simple plan shown as 8(a), and a case where after the execution of a_3 there is a disturbance that prevents executing b_1. Both systems can generate the repair in 8(b). But now consider what happens if a disturbance after a_2 makes a_3 impossible. SHOPFIXER could generate the repair in 8(c), but Höller, *et al.*'s system would regard it as incorrect, because the repaired plan tree does not include a_1 or a_2.

Bidot, Schattenberg, and Biundo (2008) present a plan repair method based on plan-modification. Their approach identifies disturbances that might break the causal dependencies in the search space of a HTN planner and produces alternatives to patch the plans. Both Bidot, Schattenberg, and Biundo and our work has been based on the ideas from (Ayan et al. 2007; Wilkins and desJardins 2001; Kambhampati and Hendler 1992). Our work uses the SHOP3 framework to generate plan repairs on the fly, so SHOPFIXER's data dependencies and repairs are incorporated in the planning algorithm: SHOPFIXER interleaves planning, plan repair, and execution using the same data structures.

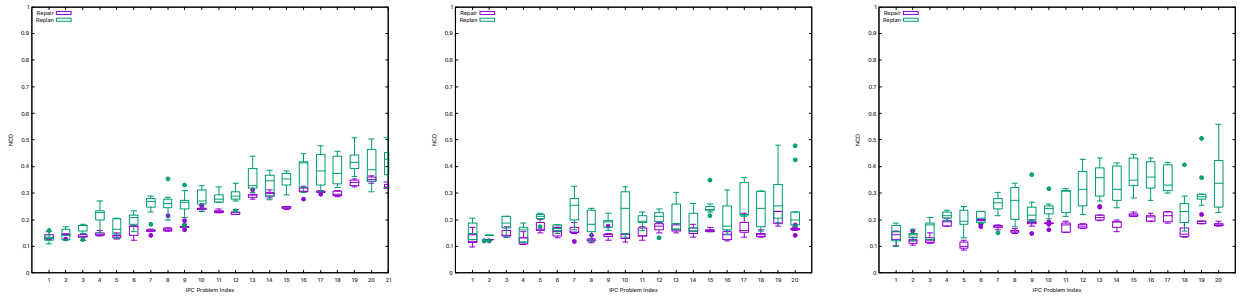
Although Bidot, Schattenberg, and Biundo's method uses the ADL-like dialect of PDDL, this seems to be limited to the primitive tasks and state representations. SHOPFIXER supports plan-repair over universally and existentially quantified expressions, increasing the HTN models that can be repaired by SHOPFIXER both theoretically and practically.

Wang and Chien describe a planning algorithm (1997) for replanning HTNs as formalized by Erol, Hendler, and Nau (1994). They extend the DPLAN algorithm (Chien et al. 1996) to replanning. Their approach is similar to HOTRiDe, but relies on the assumption that facts can be restored to their initial state when the plan fails. We did not make this assumption since it does not fit many real world domains.

8 Conclusions

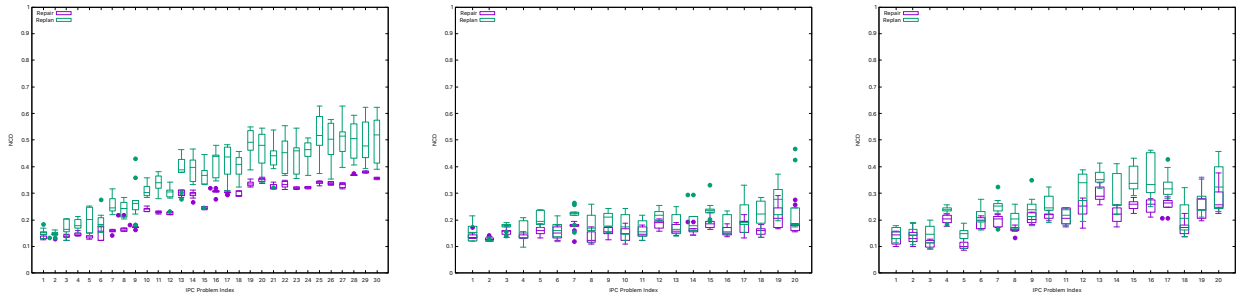
Our work addressed the issue of achieving plan stability through plan repair, as opposed to *ab initio* replanning. Our results with both SHOP3 and LPG-repair confirm, refine, and extend earlier results on repair vs. replanning from Fox et al. (2006). The one exception is that we did not universally find a computational advantage for LPG-repair over replanning: this is likely an artifact of the test domains.

Going beyond previous work, we have provided a method for minimal-perturbation replanning for SHOP3 and shown it to be sound and complete, thus going beyond previous work in this area (Ayan et al. 2007; Kuter 2012). However, while sound and complete, SHOPFIXER is built on a method



(a) Openstacks problems (b) Rovers problems (c) Satellite problems

Figure 6: SHOP3 replan NCD versus repair NCD.



(a) Openstacks problems (b) Rovers problems (c) Satellite problems

Figure 7: LPG replan NCD versus repair NCD.

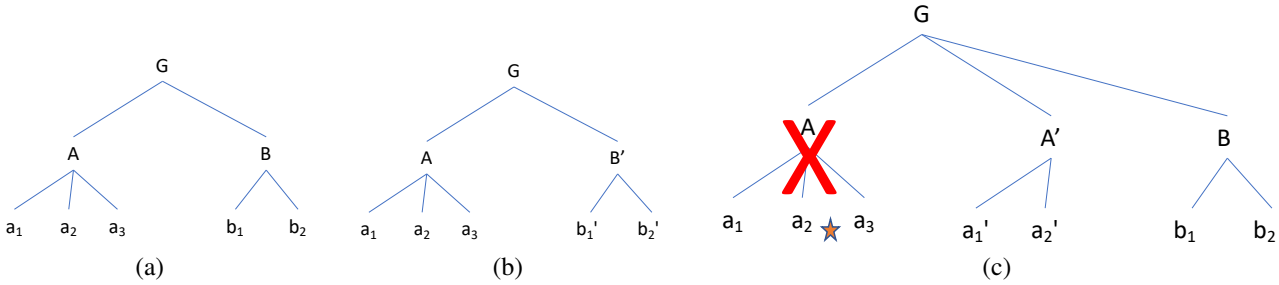


Figure 8: Höller *et al.*'s plan repair versus SHOPFIXER.

of detecting plan flaws that is complete but *unsound*, meaning that it can do extra work in some cases. We have also shown empirically that the bookkeeping overhead required by SHOPFIXER does not unduly burden planning.

In future work, we wish to extend the applicability of our techniques. SHOP3 is often used precisely because it can handle problems beyond PDDL's expressive power: the preconditions language has full Prolog expressive power, domains of quantification may not be finite, and preconditions can invoke arbitrary code. We would like to extend SHOPFIXER's expressiveness beyond the current "PDDL-like" limitations. Also, we would like to extend SHOPFIXER's stability: at present, if SHOPFIXER repairs a task T_1 generated by a method $M \rightarrow T_1 \ldots$, then the previous expansion of T_1's right siblings is lost. HOTRiDe (Ayan et al. 2007) did not have this limitation, but as mentioned earlier, used POCL planning, and is not known to be sound and complete. We are investigating *analogical replay* (Goldman et al. 2000) to retain existing plan suffixes.

A final remark: our work highlights the need for a more robust notion of "planning domain." A PDDL domain is only a way to use the same set of operators in multiple problems; it does not capture state constraints. For example, the fact that all logistics networks are fully connected, and all links are symmetric is not captured in PDDL. Hoffmann (2005) had to find these properties by empirical analysis of problem instances. Such constraints are captured only implicitly in bespoke programs that generate problems. This *substantially* complicated the process of assembling disturbance operators, and led us to rule out addressing goal changes. AI planning needs a more robust notion of domain to address issues like plan repair, planning and execution, learning, *etc.*

Acknowledgments The work reported in this paper project is sponsored by the Air Force Research Laboratory (AFRL) under contract FA8750-16-C-0182 for the Distributed Operations program. Cleared for public release, no restrictions.

References

Ayan, F.; Kuter, U.; Yaman, F.; and Goldman, R. P. 2007. HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments. In *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems – Principles and Practices for Planning in Execution*.

Bidot, J.; Schattenberg, B.; and Biundo, S. 2008. Plan repair in hybrid planning. In *Annual Conference on Artificial Intelligence*, 169–176. Springer.

Chien, S.; Govindjee, A.; Estlin, T.; Wang, X.; and Jr., R. H. 1996. Integrating hierarchical task network and operator-based planning techniques to automate operations of communications antennas.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proc. National Conf. on Artificial Intelligence (AAAI)*.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan Stability: Replanning versus Plan Repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *ICAPS*, 212–221. AAAI.

Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. San Francisco, CA: Morgan Kaufmann.

Goldman, R. P., and Kuter, U. 2015. Measuring Plan Diversity: Pathologies in Existing Approaches and A New Plan Distance Metric. In *Proceedings of the Twenty-Ninth AAAI Conference*. AAAI Press.

Goldman, R. P., and Kuter, U. 2019. Hierarchical Task Network Planning in Common Lisp: The case of SHOP3. In *Roceedings of the 12th European Lisp Symposium*.

Goldman, R. P.; Haigh, K. Z.; Musliner, D. J.; and Pelican, M. 2000. MACBeth: A Multi-Agent Constraint-Based Planner. In *Working Notes of the AAAI Workshop on Constraints and AI Planning*, 11–17.

Goldman, R. P.; Kuter, U.; and Freedman, R. G. 2020. plan-repair-icaps2020-htnws. GitHub repository with test data and experimental results. `{https://github.com/shop-planner/plan-repair-icaps2020-htnws}`.

Hoffmann, J. 2005. Where "ignoring delete lists" works: Local search topology in planning benchmarks. *JAIR* 685–758.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018. HTN plan repair using unmodified planning systems. In *Proceedings of the First ICAPS Workshop on Hierarchical Planning*, 26–30.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Kuter, U. 2012. Dynamics of behavior and acting in dynamic environments: Forethought, reaction, and plan repair. Technical Report 2012-1, SIFT.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avia, H. 1999. SHOP: Simple Hierarchical Ordered Planner. Technical Report CS-TR-3981, University of Maryland, College Park.

Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research* 20:379–404.

Schattenberg, B. 2009. *Hybrid Planning And Scheduling*. Ph.D. Dissertation, Ulm University, Institute of Artificial Intelligence. URN: urn:nbn:de:bsz:289-vts-68953.

Srivastava, B.; Nguyen, T. A.; Gerevini, A.; Kambhampati, S.; Do, M. B.; and Serina, I. 2007. Domain Independent Approaches for Finding Diverse Plans. In *Proceedings IJCAI*.

Wang, X., and Chien, S. 1997. Replanning using hierarchical task network and operator-based planning. In *Proc. European Conf. on Planning (ECP)*.

Warfield, I.; Hogg, C.; Lee-Urban, S.; and Munoz-Avila, H. 2007. Adaptation of hierarchical task network plans. In *FLAIRS-2007*.

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.