Coming up With Good Excuses: What to do When no Plan Can be Found

Moritz Göbelbecker and **Thomas Keller**
and **Patrick Eyerich** and **Michael Brenner** and **Bernhard Nebel**
University of Freiburg, Germany
{goebelbe, tkeller, eyerich, brenner, nebel}@informatik.uni-freiburg.de

Abstract

When using a planner-based agent architecture, many things can go wrong. First and foremost, an agent might fail to execute one of the planned actions for some reasons. Even more annoying, however, is a situation where the agent is *incompetent*, i.e., unable to come up with a plan. This might be due to the fact that there are principal reasons that prohibit a successful plan or simply because the task's description is incomplete or incorrect. In either case, an explanation for such a failure would be very helpful. We will address this problem and provide a formalization of *coming up with excuses* for not being able to find a plan. Based on that, we will present an algorithm that is able to find excuses and demonstrate that such excuses can be found in practical settings in reasonable time.

Introduction

Using a planner-based agent architecture has the advantage that the agent can cope with many different situations and goals in flexible ways. However, there is always the possibility that something goes wrong. For instance, the agent might fail to execute a planned action. This may happen because the environment has changed or because the agent is not perfect. In any case, recovering from such a situation by recognizing the failure followed by replanning is usually possible (Brenner and Nebel 2009).

Much more annoying than an execution failure is a failure to find a plan. Imagine a household robot located in the living room with a locked door to the kitchen that receives the order to tidy up the kitchen table but is unable to come up with a plan. Better than merely admitting it is *incompetent* would be if the robot could provide a good *excuse* – an explanation of *why* it was not able to find a plan. For example, the robot might recognize that if the kitchen door were unlocked it could achieve its goals.

In general, we will adopt the view that an excuse is a counterfactual statement (Lewis 1973) of the form that a small change of the planning task would permit the agent to find a plan. Such a statement is useful when debugging a domain description because it points to possible culprits that prevent finding a plan. Also in a regular setting a counterfactual explanation is useful because it provides a hint for where to start when trying to resolve the problem, e.g., by asking for help from a human or exploring the space of possible repair actions.

There are many ways to change a planning task so that it becomes possible to generate a plan. One may change

- the goal description,
- the initial state, or
- the set of planning operators.

Obviously, some changes are more reasonable than others. For example, weakening the goal formula is, of course, a possible way to go. We would then reduce the search for excuses to over-subscription planning (Smith 2004). However, simply ignoring goals would usually not be considered as an excuse or explanation.

On the other hand, changing the initial state appears to be reasonable, provided we do not make the trivial change of making goal atoms true. In the household robot example, changing the state of the door would lead to a solvable task and thus give the robot the possibility to actually realize the reasons of its inability to find a plan.

In some cases, it also makes sense to add new objects to the planning task, e.g., while the robot is still missing sensory information about part of its environment. Thus, we will consider changes to the object domain as potential changes of the task, too. Note that there are also situations in which removing objects is the only change to the initial state that may make the problem solvable. However, since these situations can almost always be captured by changing those objects' properties, we ignore this special case in the following.

Finally, changing the set of planning operators may indeed be a "better" way, e.g., if an operator to unlock the door is missing. However, because the number of potential changes to the set of operators exceeds the number of changes to the initial state by far, we will concentrate on the latter in the remainder of the paper, which also seems like the most intuitive type of explanation.

The rest of the paper is structured as follows. In the next section, we introduce the formalization of the planning framework we employ. After that we sketch a small motivating example. Based on that, we will formalize the notion of excuses and determine the computational complexity of finding excuses. On the practical side, we present a method

that is able to find good excuses, followed by a section that shows our method's feasibility by presenting empirical data on some IPC planning domains and on domains we have used in a robotic context. Finally, we discuss related work and conclude.

The Planning Framework

The planning framework we use in this paper is the ADL fragment of PDDL2.2 (Edelkamp and Hoffmann 2004) extended by multi-valued fluents as in the SAS$^+$ formalism (Bäckström and Nebel 1995) or functional STRIPS (Geffner 2000).[1] One reason for this extension is that modeling using multi-valued fluents is more intuitive. More importantly, changing fluent values when looking for excuses leads to more intuitive results than changing truth values of Boolean state variables, since we avoid states that violate implicit domain constraints. For example, if we represent the location of an object using a binary predicate $at(\cdot,\cdot)$, changing the truth value of a ground atom would often lead to having an object at two locations simultaneously or nowhere at all. The domain description does not tell us that, if we make a ground atom with the *at* predicate true, we have to make another ground atom with the identical first parameter false. By using multi-valued fluents instead, such implicit constraints are satisfied automatically.

Of course, our framework can be applied to any reasonable planning formalism, since it is simply a matter of convenience to have multi-valued fluents in the language. This being said, a **planning domain** is a tuple $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$, where

- $\mathcal{T}$ are the **types**,
- $\mathcal{C}_\Delta$ is the set of **domain constant symbols**,
- $\mathcal{S}$ is the set of **fluent** and **predicate symbols** with associated arities and typing schemata, and
- $\mathcal{O}$ is the set of **planning operators** consisting of preconditions and effects.

A **planning task** is then a tuple $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, where

- Δ is a planning domain as defined above,
- $\mathcal{C}_\Pi$ is a set of **task-dependent constant symbols** disjoint from $\mathcal{C}_\Delta$,
- s_0 is the description of the **initial state**, and
- s^* is the **goal** specification.

The initial state is specified by providing a set s_0 of ground atoms, e.g., `(holding block1)` and ground fluent assignments, e.g., `(= (loc obj1) loc2)`. As usual, the description of the initial state is interpreted under the *closed world assumption*, i.e., any logical ground atom not mentioned in s_0 is assumed to be false and any fluent not mentioned is assumed to have an undefined value. In the following sections we assume that $\mathcal{S}$ contains only fluents and no predicates at all. More precisely, we will treat predicates as fluents with a domain of $\{\bot, \top\}$ and a default value of $\bot$ instead of unknown.

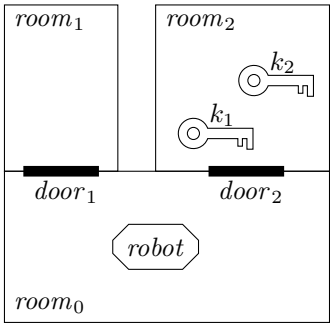
The goal specification is a closed first-order formula over logical atoms and fluent equalities. We say that a planning task is **solvable** iff there is a plan Ψ that transforms the state described by s_0 into a state that satisfies the goal specification s^*.

Sometimes we want to turn an initial state description into a (sub-)goal specification. Assuming that plan Ψ solves the task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, by $nec(s_0, \Psi, s^*)$ we mean the formula that describes the setting of the fluents necessary for the correct execution of Ψ started in initial state s_0 leading to a state satisfying s^*. Note that $s_0 \models nec(s_0, \Psi, s^*)$ always holds.

Motivating Examples

The motivation for the work described in this paper mostly originated from the DESIRE project (Plöger et al. 2008), in which a household robot was developed that uses a domain-independent planning system. More often than not a sensor did not work the way it was supposed to, e.g., the vision component failed to detect an object on a table. If the user-given goal is only reachable by utilizing the missing object, the planning system naturally cannot find a plan. Obviously, thinking ahead of everything that might go wrong in a real-life environment is almost impossible, and if a domain-independent planning system is used, it is desirable to also realize flaws in a planning task with domain-independent methods. Furthermore, we wanted the robot to not only realize that something went wrong (which is not very hard to do after all), but it should also be able to tell the user what went wrong and why it couldn't execute a given command.

However, not only missing objects may cause problems for a planning system. Consider a simple planning task on the KEYS-domain, where a robot navigates between rooms through doors that can be unlocked by the robot if it has the respective key (see Fig. 1). The goal of such a task is to have the robot reach a certain room, which in this example is $room_1$.




```
(:objects
  room0 room1 room2 - room
  door1 door2 - door
  key1 key2 - key)

(:init
  (= (robot_pos) room0)
  (= (key_pos key1) room2)
  (= (key_pos key2) room1)
  (connects room0 room1 door1)
  (connects room1 room0 door1)
  (connects room0 room2 door2)
  (connects room2 room0 door2)
  (= (key_opens key1) door1)
  (= (key_opens key2) door2)
  (locked door1)
  (locked door2))

(:goal (= (robot_pos) room1))
```

Figure 1: Unsolvable task in the *Keys* domain with corresponding PDDL code.

Obviously there exists no plan for making the robot reach its goal. What, however, are the *reasons* for this task being

[1] Multi-valued fluents have been introduced to PDDL in version 3.1 under the name of "object fluents".

unsolvable? As we argued in the introduction, the answer to this question can be expressed as a counterfactual statement concerning the planning task. Of course, there are numerous ways to change the given problem such that a solution exists, the easiest one certainly being to already have the goal fulfilled in the initial state. An only slightly more complicated change would be to have the door to $room_1$ state) in which case the robot could directly move to its destination, or have the robot already carry the key to that door (changing the value of `(key_pos key1)` from `room2` to `robot`) or even a new one (adding an object of type key with the required properties), or simply have one of the keys in $room_0$ (e.g. `(= (key_pos key1) room0)`).

Having multiple possible excuses is, as in this case, rather the rule than the exception, and some of them are more reasonable than others. So, the following sections will answer two important questions. Given an unsolvable planning task, *What is a good excuse?* and *How to find a good excuse?*

Excuses

As spelled out above, for us an excuse is a change in the initial state (including the set of objects) with some important restrictions: We disallow the deletion of objects and changes to any fluents that could contribute to the goal. For example, we may not change the location of the robot if having the robot at a certain place is part of the goal.

A ground fluent f **contributes** to the goal if adding or deleting an assignment $f = x$ from a planning state can make the goal true. Formally, f contributes to s^* iff there exists a state s with $s \not\models s^*$ such that $s \cup \{f = x\} \models s^*$ for some value x.

Given an unsolvable planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, an **excuse** is a tuple $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ that implies the solvable **excuse task** $\Pi^\chi = \langle \Delta, \mathcal{C}_\chi, s_\chi, s^* \rangle$ such that $\mathcal{C}_\Pi \subseteq \mathcal{C}_\chi$ and if $(f = x) \in s_0 \triangle s_\chi$ (where $\triangle$ denotes the symmetric set difference) then f must not contribute to s^*.

The changed initial state s_χ is also called **excuse state**.

It should be noted that it is possible that no excuse exists, e.g., if there is no initial state such that a goal state is reachable. More precisely, there is no excuse iff the task is solvable or all changes to the initial state that respect the above mentioned restrictions do not lead to a solvable task.

Acceptable Excuses

If we have two excuses and one changes more initial facts than the other, it would not be an acceptable excuse, e.g., in our example above moving both keys to the room where the robot is would be an excuse. Relocating any one of them to that room would already suffice, though.

So, given two excuses $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ and $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle$, we say that χ is at least as **acceptable** as χ', written $\chi \preceq \chi'$, iff $\mathcal{C}_\chi \subseteq \mathcal{C}_{\chi'}$ and $s_0 \triangle s_\chi \subseteq s_0 \triangle s_{\chi'}$. A minimal element under the ordering $\preceq$ is called an **acceptable excuse**.

Good Excuses

Given two acceptable excuses, it might nevertheless be the case that one of them subsumes the other if the changes in one excuse can be explained by the other one.

In the example from Fig. 1, one obvious excuse χ would lead to a task in which $door_1$ was unlocked so that the robot could enter $room_1$. This excuse, however, is unsatisfactory since the robot itself could unlock $door_1$ if its key was located in $room_0$ or if $door_2$ was unlocked. So any excuse χ' that contains one of these changes should subsume χ.

We can formalize this subsumption as follows: Let $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ be an acceptable excuse to a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ with the plan Ψ solving Π^χ. Another acceptable excuse $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle$ to Π is called **at least as good as** χ, in symbols $\chi' \sqsubseteq \chi$, if χ' is an acceptable excuse also to $\langle \Delta, \mathcal{C}_\Pi, s_0, \mathit{nec}(s_{\chi'}, \Psi, s^*) \rangle$. We call χ' better than χ, in symbols $\chi' \sqsubset \chi$, iff $\chi' \sqsubseteq \chi$ and $\chi \not\sqsubseteq \chi'$.

In general, good excuses would be expected to consist of changes to so-called *static facts*, facts that cannot be changed by the planner and thus cannot be further regressed from, as captured by the above definition. In our example this could be a new key – with certain properties – and perhaps some additional unlocked doors between rooms.

However, there is also the possibility that there are cyclic dependencies as in the children's song *There's a Hole in the Bucket*. In our example, one excuse would be χ, where the door to $room_2$ is unlocked. In a second one, χ', the robot carries key k_2. Obviously, $\chi \sqsubseteq \chi'$ and $\chi' \sqsubseteq \chi$ hold and thus χ and χ' form a cycle in which all excuses are equally good.

In cases with cyclic dependencies, it is still possible to find even "better" excuses by introducing additional objects, e.g., a new door or a new key in our example. However, cyclic excuses as above, consisting of χ and χ', appear to be at least as intuitive as excuses with additional objects. For these reasons, we define a **good** excuse χ as one such that there either is no better excuse or there exists a different excuse χ' such that $\chi \sqsubseteq \chi'$ and $\chi' \sqsubseteq \chi$.

Perfect Excuses

Of course, there can be many good excuses for a task, and one may want to distinguish between them. A natural way to do so is to introduce a cost function that describes the cost to transform one state into another. Of course, such a cost function is just an estimate because the planner has no way to transform the initial state into the excuse state.

Here, we will use a cost function $c(\cdot)$, which should respect the above mentioned acceptability ordering $\preceq$ as a minimal requirement. So, if $\chi' \preceq \chi$, we require that $c(\chi') \leq c(\chi)$. As a simplifying assumption, we will only consider additive cost functions. So, all ground fluents have predefined costs, and the cost of an excuse is simply the sum over the costs of all facts in the symmetric difference between initial and excuse state. Good excuses with minimal costs are called **perfect excuses**.

Computational Complexity

In the following, we consider ordinary propositional and DATALOG planning, for which the problem of deciding plan existence – the PLANEX problem – is PSPACE- and EXPSPACE-complete, respectively (Erol, Nau, and Subrahmanian 1995). In the context of finding excuses, we will mainly consider the following problem for acceptable, good, and perfect excuses:

- EXCUSE-EXIST: Does there exist any excuse at all?

In its unrestricted form, this problems is undecidable for DATALOG planning.

Theorem 1 *EXCUSE-EXIST is undecidable for* DATALOG *planning.*

Proof Sketch. The main idea is to allow an excuse to introduce an unlimited number of new objects, which are arranged as tape cells of a Turing machine. That these tape cells are empty and have the right structure could be verified by an operator that must be executed in the beginning. After that a Turing machine could be simulated using ideas as in Bylander's proof (Bylander 1994). This implies that the Halting problem on the empty tape can be reduced to EXCUSE-EXIST, which means that the latter is undecidable. ∎

However, an excuse with an unlimited number of new objects is, of course, also not very intuitive. For these reasons, we will only consider BOUNDED-EXCUSE-EXIST, where only a polynomial number of new objects is permitted. As it turns out, this version of the problem is not more difficult than planning.

Lemma 2 *There is a polynomial Turing reduction from PLANEX to BOUNDED-EXCUSE-EXIST for acceptable, good, or perfect excuses.*

Proof Sketch. Given a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ with planning domain $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$, construct two new tasks by extending the set of predicates in $\mathcal{S}$ by a fresh ground atom a leading to $\mathcal{S}'$. In addition, this atom is added to all preconditions in the set of operators resulting in $\mathcal{O}'$. Now we generate:

$$\begin{aligned} \Pi' &= \langle \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}', \mathcal{O}' \rangle, \mathcal{C}_\Pi, s_0, s^* \rangle \\ \Pi'' &= \langle \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}', \mathcal{O}' \rangle, \mathcal{C}_\Pi, s_0 \cup \{a\}, s^* \rangle \end{aligned}$$

Obviously, Π is solvable iff there exists an excuse for Π' and there is no excuse for Π''. ∎

It is also possible to reduce the problems the other way around, provided the planning problems are in a deterministic space class.

Lemma 3 *The BOUNDED-EXCUSE-EXIST problem can be Turing reduced to the PLANEX problem – provided PLANEX is complete for a space class that includes PSPACE.*

Proof Sketch. By Savitch's theorem (1980), we know that NSPACE$(f(n)) \subseteq$ DSPACE$((f(n))^2)$, i.e., that all deterministic space classes including PSPACE are equivalent to their non-deterministic counterparts. This is the main reason why finding excuses is not harder than planning.

Let us assume that the plan existence problem for our formalism is XSPACE-complete. Given a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, the following algorithm will determine whether there is an excuse:

1. If Π is solvable, return "no".
2. Guess a $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ and verify the following:
   a) $\mathcal{C}_\Pi \subseteq \mathcal{C}_\chi$;
   b) $\Pi^\chi = \langle \Delta, \mathcal{C}_\chi, s_\chi, s^* \rangle$ is solvable;

This non-deterministic algorithm obviously needs only XSPACE using an XSPACE-oracle for the PLANEX problem. Since the existence of an excuse implies that there is a perfect excuse (there are only finitely many different possible initial states), the algorithm works for all types of excuses. ∎

From the two lemmas, it follows immediately that the EXCUSE-EXIST problem and the PLANEX problem have the same computational complexity.

Theorem 4 *The BOUNDED-EXCUSE-EXIST problem is complete for the same complexity class as the PLANEX problem for all planning formalisms having a PLANEX problem that is complete for a space class containing PSPACE.*

Using similar arguments, it can be shown that we can compute which literals in the initial state can be relevant or are necessary for an excuse. By guessing and verifying using PLANEX-oracles, these problems can be solved and are therefore in the same space class as the PLANEX problems, provided they are complete for a space class including PSPACE.

Candidates for Good Excuses

The range of changes that may occur in acceptable excuses is quite broad: The only excuses forbidden are those that immediately contribute to the goal. We could try to find acceptable excuses and apply goal regression until we find a good excuse, but this would be highly suboptimal, because it might require a lot of goal regression steps. Therefore, we first want to explore if there are any constraints (on fluent or predicate symbols, source or target values) that must be satisfied in any *good* excuse.

In order to analyze the relations between fluent symbols, we apply the notion of *causal graphs* and *domain transition graphs* (Helmert 2006) to the abstract domain description.

The **causal graph** CG_Δ of a planning domain $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$ is a directed graph $(\mathcal{S}, A)$ with an arc $(u, v) \in A$ if there exists an operator $o \in \mathcal{O}$ so that $u \in \mathbf{pre}(o)$ and $v \in \mathbf{eff}(o)$ or both u and v occur in $\mathbf{eff}(o)$. If $u = v$ then (u, v) is in A iff the fluents in the precondition and effect can refer to distinct instances of the fluent.

The causal graph captures the dependencies of fluents on each other; to analyze the ways the values of one fluent can change, we build its domain transition graph. In contrast to the usual definition of domain transition graphs (which is based on grounded planning tasks), the domain of a fluent f can consist of constants as well as free variables. This fact needs to be taken into account when making statements about the domain transition graph (e.g., the reachability of a variable of type t implies the reachability of all variables of subtypes of t). For the sake of clarity, we will largely gloss over this distinction here and treat the graph like its grounded counterpart.

If $dom(f)$ is the domain of a fluent symbol $f \in \mathcal{S}$, its **domain transition graph** $\mathcal{G}_f$ is a labeled directed graph

$(dom(f), E)$ with an arc $(u, v) \in E$ iff there is an operator $o \in \mathcal{O}$ so that $f = u$ is contained in $\mathbf{pre}(o)$ and $f = v \in \mathbf{eff}(o)$. The label consists of the preconditions of o minus the precondition $f = u$. An arc from the unknown symbol, $(\bot, v)$, exists if f does not occur in the precondition. We also call such an arc $(u, v) \in \mathcal{G}_f$ a **transition** of f from u to v and the label of (u, v) its precondition.

For example, the domain transition graph of the `robot_pos` fluent has one vertex consisting of a variable of type `room` and one edge $(\texttt{room}, \texttt{room})$ with the label of $\texttt{connected}(\mathit{room_1}, \mathit{room_2}, \mathit{door}) \wedge \texttt{open}(\mathit{door})$.

In order to constrain the set of possible excuses, we restrict candidates to those fluents and values that are relevant for achieving the goal. The **relevant domain**, $dom_{rel}(f)$, of a fluent f is defined by the following two conditions and can be calculated using a fixpoint iteration: If $f = v$ contributes to the goal, then $v \in dom_{rel}(f)$. Furthermore, for each fluent f' on which f depends, $dom_{rel}(f')$ contains the subset of $dom(f')$ which is (potentially) required to reach any element of $dom_{rel}(f)$.

A **static change** is a change for which there is no path in the domain-transition graph even if all labels are ignored. Obviously all changes to *static variables* are static, but the converse is not always true. For example, in most planning domains, if in a planning task a non-static fluent f is undefined, setting f to some value x would be a static change.

In the following, we show that in some cases it is sufficient to consider static changes as candidates for excuses in order to find all good excuses. To describe these cases, we define two criteria, mutex-freeness and strong connectedness, that must hold for static and non-static fluents, respectively.

We call a fluent f **mutex-free** iff changing the value of an instance of f in order to enable a particular transition of a fluent f' that depends on f does not prevent any other transition. Roughly speaking, excuses involving f' are not good, because they can always be regressed to the dependencies f without breaking anything else. Two special cases of mutex-free fluents are noteworthy, as they occur frequently and can easily be found by analyzing the domain description: If a fluent f is *single-valued*, i.e. there are no two operators which depend on different values for f, it is obviously mutex-free. A less obvious case is free variables. Let o be an operator that changes the fluent $f(p_1, \ldots p_n)$ from p_v to p'_v. A precondition $f'(q_1, \ldots q_n) = q_v$ of o has free variables iff there is at least one variable in $q_1, \ldots, q_n$ that doesn't occur in $\{p_1, \ldots p_n, p_v, p'_v\}$. Here the mutex-freeness is provided because we can freely add new objects to the planning task and thus get new grounded fluents that cannot interfere with any existing fluents.

For example, consider the `unlock` operator in the KEYS-domain. Its precondition includes $\texttt{key\_pos}(\mathit{key}) = \texttt{robot} \wedge \texttt{key\_opens}(\mathit{key}) = \mathit{door}$. Here *key* is a free variable, so it is always possible to satisfy this part of the precondition by introducing a new key object and setting its position to the robot and its `key_opens` property to the door we want to open. As we do not have to modify an existing key, all actions that were previously applicable remain so.

The second criterion is the connectedness of the domain transition graph. We call a fluent f **strongly connected** iff the subgraph of $\mathcal{G}_f$ induced by the relevant domain of f is strongly connected. This means that once f has a value in $dom_{rel}(f)$ any value that may be relevant for achieving the goal can be reached in principle. In practice, most fluents have this property because any operator that changes a fluent from one free variable to another connects all elements of that variable's type.

Theorem 5 *Let Δ be a domain with an acyclic causal graph where all non-static fluents are strongly connected and all static fluents are mutex-free.*

Then any good excuse will only contain static changes.

Proof. First note that a cycle free causal graph implies that there are no co-occurring effects, as those would cause a cycle between their fluents.

If an excuse $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ for $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ is an excuse that contains non-static changes, then we will construct an excuse $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle \sqsubset \chi$ containing only static facts which can explain χ. As all static fluents are mutex-free, no changes made to the initial state $s_{\chi'}$ to fulfill static preconditions can conflict with changes already made to s_χ, so we can choose the static changes in χ' to be those that make all (relevant) static preconditions true.

Let f, v, v' be a non-static change, i.e., $f = v \in s_0$ and $f = v' \in s_\chi$. This means that there exists a path from v to v' in $\mathcal{G}_f$. If all preconditions along this path are static, we are done as all static preconditions are satisfied in $s_{\chi'}$. If there are non-static preconditions along the path from v to v', we can apply this concept recursively to the fluents of those preconditions. As there are no co-occurring effects and the relevant part of each non-static fluent's domain transition graph is strongly connected, we can achieve all preconditions for each action and restore the original state later. ∎

We can easily extend this result to domains with a cyclic causal graph:

Theorem 6 *Let Δ be a domain where all non-static fluents are strongly connected, all static fluents are mutex-free and each cycle in the domain's causal graph contains at least one mutex-free fluent.*

Then any good excuse will only contain static changes or changes that involve a fluent on a cycle.

Proof. We can reduce this case to the non-cyclic case, by removing all effects that change the fluents f fulfilling the mutex-free condition, thus making them static. Let us call this modified domain Δ'.

Let χ be an excuse with non-static changes. Because Δ' contains only a subset of operators of Δ, any non-static change that can further be explained in Δ' can also be explained in Δ. So there exists an $\chi' \sqsubset \chi$, which means that χ cannot be a good excuse unless the changed fluent lies on a cycle so that $\chi \sqsubset \chi'$ may hold, too. ∎

While these conditions may not apply to all common planning domains as a whole, they usually apply to a large enough set of the fluents so that limiting the search to static and cyclic excuses speeds up the search for excuses significantly without a big trade-off in optimality.

Finding Excuses Using a Cost-Optimal Planner

We use the results from the previous section to transform the problem of finding excuses into a planning problem by adding operators that change those fluents that are candidates for good excuses. If we make sure that those **change operators** can only occur at the start of a plan, we get an excuse state s_χ by applying them to s_0.

Given an (unsolvable) planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, we create a transformed task with action costs $\Pi' = \langle \Delta', \mathcal{C}_{\Pi'}, {s_0}', s^* \rangle$ as follows.

We recursively generate the relevant domain for each fluent symbol $f \in \mathcal{S}$ by traversing the causal graph, starting with the goal symbols. During this process, we also identify cyclic dependencies. Then we check $\mathcal{G}_f$ for reachability, adding all elements of $dom(f)$ from which $dom_{rel}(f)$ is **not** reachable to $changes(f)$. If f is involved in a cyclic dependency we also add $dom_{rel}(f)$ to $changes(f)$.

To prevent further changes to the planning state after the first execution of a regular action, we add the predicate $started$ to $\mathcal{S}'$ and as a positive literal to the effects of all operators $o \in \mathcal{O}$.

For every fluent f (with arity n) and $v \in changes(f)$ we introduce a new operator set_v^f as follows:

$$\mathbf{pre}(set_v^f) = \neg started \ \wedge f(p_1 \dots p_n) = v \bigwedge_{i=1}^{n} \neg unused(p_i)$$

$$\mathbf{eff}(set_v^f) = \{f(p_1 \dots p_n) = p_{n+1}\}$$

To add a new object of type t to the initial state, we add a number[2] of **spare objects** $sp_1^t \dots sp_n^t$ to $\mathcal{C}_{\Pi'}$. For each of these objects sp_i^t, the initial state ${s_0}'$ contains the facts $unused(sp_i^t)$. We then add the operator $add^t(p)$ with $\mathbf{pre}(add^t) = unused(p) \wedge \neg started$ and $\mathbf{eff}(add^t) = \{\neg unused(p)\}$.

To prevent the use of objects that have not been activated yet we add $\neg unused(p_i)$ to each operator $o \in \mathcal{O}$ for each parameter p_i to $pre(o)$ if $pre(o)$ does not contain a fluent or positive literal with p_i as parameter.

Due to the use of the $started$ predicate, any plan Ψ can be partitioned into the actions before $started$ was set (those that change the initial state) and those after. We call the subplans Ψ_{s_0} and Ψ_Π, respectively.

As a final step we need to set the costs of the change actions. In this implementation we assume an additive cost function that assigns non-zero costs to each change and does not distinguish between different instances of a fluent, so $c(f)$ are the costs associated with changing the fluent f and $c(t)$ the costs of adding an object of type t. We set $c(set_v^f) = \alpha c(f)$ and $c(add^t) = \alpha c(t)$ with α being a scaling constant. We need to make sure that the costs of the change actions always dominate the total plan's costs as otherwise worse excuses might be found if they cause Ψ_Π to be shorter. We can achieve this by setting α to an appropriate upper bound of the plan length in the original problem Π.

From the resulting plan Ψ we can easily construct an excuse $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ with $\mathcal{C}_\Psi = \mathcal{C}_\Pi \cup \{c : add^t(c) \in \Psi\}$ and s_Ψ being the state resulting from the execution of Ψ_{s_0} restricted to the fluents defined in the original Problem Π.

Theorem 7 *Let Π be a planning task, Ψ an optimal solution to the transformed task Π', and $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ the excuse constructed from Ψ. Then χ is an acceptable excuse to Π.*

Proof. Ψ_Π only contains operators in Δ and constants from $\mathcal{C}_\chi$. Obviously Ψ_Π also reaches the goal from s_Ψ. So Π^χ is solvable and χ thus an excuse. To show that χ is acceptable, we need to show that no excuse with a subset of changes exists. If such an excuse χ' existed it could be reached by applying change operators (as the changes in χ' are a subset of those in χ). Then a plan Ψ' would exist with $c(\Psi'_{s_0}) < c(\Psi_{s_0})$ and, as the cost of Ψ_{s_0} always dominates the cost of Ψ_Π, $c(\Psi') < c(\Psi)$. This contradicts that Ψ is optimal, so χ must be acceptable. ∎

Theorem 8 *Let Π be a planning task, Ψ an optimal solution to the transformed task Π', and $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ the excuse constructed from Ψ. If χ changes only static facts, it is a perfect excuse.*

Proof. As χ contains only static facts, it must be a good excuse. From the definition of the cost function it follows that $c(\chi) = \alpha c(\Psi_{s_0})$, so existence of an excuse χ' with $c(\chi') < c(\chi)$ would imply, as in the previous proof, the existence of a plan Ψ' with $c(\Psi') < c(\Psi)$, contradicting the assumption that Ψ is optimal. ∎

Cyclic Excuses

Solving the optimal planning problem will not necessarily give us a good excuse (unless the problem's causal graph is non-cyclic, of course). So if we get an excuse that changes a non-static fact, we perform a *goal regression* as described earlier. We terminate this regression when all new excuses have already been encountered in previous iterations or no excuse can be found anymore. In the former case we select the excuse with the lowest cost from the cycle, in the latter case we need to choose the last found excuse.

Note though, that this procedure will not necessarily find excuses with globally optimal costs: As there is no guarantee that $\chi' \sqsubset \chi$ also implies $c(\chi') \leq c(\chi)$ the goal regression might find excuses that have higher costs than a good excuse that might be found from the initial task Π.

Experiments

To test our implementation's quality we converted selected planning tasks of the IPC domains LOGISTICS (IPC'00), ROVERS and STORAGE (both IPC'06) to use object fluents, so that our algorithm could work directly on each problem's SAS^+ representation. In order to give our program a reason to actually search for excuses, it was necessary to create flaws in each problem's description that made it unsolvable. For each problem file, we modified the initial state by randomly deleting any number of valid fluents and predicates,

[2] As shown in the complexity discussion, the number of new objects might be unreasonably high. In some cases this number can be restricted further but this has been left out for space reasons. In practice we cap the number of spares per type with a small constant.

	sat 0	**opt** 0	**sat** 1	**opt** 1	**sat** 2	**opt** 2	**sat** 3	**opt** 3	**sat** 4	**opt** 4
logistics-04	0.78s	1.43s	0.69s (0.5)	0.94s (0.5)	0.71s (1.5)	1.02s (1.5)	0.53s (1.0)	0.57s (1.0)	0.52s (2.5)	1.29s (2.5)
logistics-06	0.75s	9.81s	0.74s (1.5)	28.12s (1.5)	0.65s (2.5)	101.47s (2.5)	0.65s (3.0)	55.05s (2.5)	0.62s (3.5)	43.57s (3.5)
logistics-08	1.27s	76.80s	1.27s (1.0)	276.99s (1.0)	1.17s (1.0)	46.47s (1.0)	1.08s (5.5)	1176.49s (3.5)	0.96s (5.5)	1759.87s (4.5)
logistics-10	2.62s	—	2.24s (2.0)	—	2.36s (5.5)	—	2.25s (4.0)	—	1.29s (5.5)	—
logistics-12	2.58s	—	2.66s (2.0)	—	2.66s (4.5)	—	2.28s (5.0)	—	1.89s (6.5)	—
logistics-14	4.73s	—	4.78s (2.5)	—	4.24s (6.0)	—	3.70s (7.5)	—	2.71s (6.0)	—
rovers-01	3.04s	3.61s	3.09s (0.5)	5.72s (0.5)	3.17s (1.5)	8.17s (1.5)	2.79s (5.5)	—	2.90s (7.5)	—
rovers-02	3.25s	3.79s	3.24s (0.5)	4.45s (0.5)	3.31s (2.5)	21.48s (2.5)	3.23s (3.0)	62.36s (3.0)	2.87s (6.5)	—
rovers-03	4.15s	5.53s	4.11s (0.5)	7.90s (0.5)	3.55s (2.5)	112.43s (2.5)	4.04s (5.5)	—	3.67s (6.5)	—
rovers-04	5.01s	6.53s	4.94s (1.0)	8.97s (0.5)	68.60s (5.0)	22.01s (2.0)	3.21s (6.0)	—	9.45s (12.0)	—
rovers-05	5.29s	—	6.23s (2.0)	925.61s (2.0)	7.25s (4.0)	—	5.82s (5.0)	790.57s (5.0)	6.32s (8.0)	—
storage-01	1.77s	1.83s	2.01s (0.5)	2.31s (0.5)	1.71s (3.0)	2.11s (2.0)	1.84s (5.0)	24.81s (4.0)	1.82s (4.5)	11.12s (3.5)
storage-05	11.14s	15.66s	10.85s (0.5)	37.09s (0.5)	8.25s (4.0)	53.38s (4.0)	10.25s (6.0)	—	31.70s (6.0)	—
storage-08	30.46s	101.32s	35.59s (1.5)	—	774.17s (5.5)	—	765.32s (7.5)	—	110.31s (8.5)	—
storage-10	88.07s	214.10s	62.93s (1.0)	—	64.56s (2.0)	—	423.71s (3.0)	—	257.10s (4.0)	—
storage-12	131.36s	—	—	—	—	—	—	—	—	—
storage-15	1383.65s	—	—	—	—	—	—	—	—	—

Table 1: Results for finding excuses on some IPC domains. All experiments were conducted on a 2.66 GHz Intel Xeon processor with a 30 minutes timeout and a 2 GB memory limit. We used two setting for the underlying Fast Downward Planner: **sat** is Weighted A* with the enhanced-additive heuristic and a weight of 5, **opt** is A* with the admissible LM Cut Heuristic. For each problem instance there are five versions: the original (solvable) version is referred to as 0 while versions 1 to 4 are generated according to the deletions described in the Experiments section. We used an uniform cost measure with the exception that assigning a value to a previously undefined fluent costs 0.5. Runtime results are in seconds; the excuses costs are shown in parentheses.

or by completely deleting one or more objects necessary to reach the goal in every possible plan (the latter includes the deletion of all fluents and predicates containing the deleted object as a parameter). For instance, in the LOGISTICS domain, we either deleted one or more *city-of* fluents, or all trucks located in the same city, or all airplanes present in the problem.

In order to not only test on problems that vary in the difficulty to find a plan, but also the difficulty to find excuses, we repeated this process four times, each repetition taking the problem gained in the iteration before as the starting point. This lead to four versions of each planning task, each one missing more initial facts compared to the original task than the one before.

Our implementation is based on the Fast Downward planning system (Helmert 2006), using a Weighted A^* search with an extended context-enhanced additive heuristic that takes action costs into account. Depicted are runtimes and the cost of the excuse found. Because this heuristic is not admissible, the results are not guaranteed to be optimal, so we additionally ran tests using A^* and the (admissible) landmark cut heuristics (Helmert and Domshlak 2009).

To judge the quality of the excuses produced we used a uniform cost measure, with one exception: The cost of the assignment of a concrete value to a previously undefined fluent is set to be 0.5. This kind of definition captures our definition of acceptable excuses via the symmetric set difference and also appears to be natural: Assigning a value to an undefined fluent should be of lower cost than changing a value that was given in the original task. Note that switching a fluent's value actually has a cost of 1.0.

As the results in Table 1 show, the time for finding excuses increases significantly in the larger problems. The principal reason for that is that previously static predicates like `connected` in the STORAGE domain have become non-static due to the introduction of change operators. This leads both to a much larger planning state (as they cannot be compiled away anymore), as well as a much larger amount of applicable ground actions. This effect can be seen in the first two columns which show the planning times on the unmodified problems (but with the added change operators).

As expected, optimal search was able to find excuses for fewer problems than satisficing search. Satisficing search came up with excuses for most problems in a few seconds with the exception of the storage domain, due to the many static predicates. The costs of the excuses found were sometimes worse than those found by optimal search, but usually not by a huge amount. If better excuses are desired, additional tests showed that using smaller weights for the Weighted A^* are a reasonable compromise.

It is interesting to note that for the satisficing planner the number of changes needed to get a solvable task has little impact on the planning time. The optimal search, on the other hand, usually takes much longer for the more flawed problems. A possible explanation for this behavior is that the number of possible acceptable excuses grows drastically the more facts we remove from the problem. This makes finding *some* excuse little harder, but greatly increases the difficulty of finding an *optimal* excuse.

While most of the excuses described in this paper can be found in the problems we created this way, it is very unlikely that a problem is contained that is unsolvable due to a cyclic excuse[3]. The aforementioned KEYS-domain on the other hand is predestined to easily create problems that are unsolvable because of some cyclic excuse. So for our second experiment, we designed problems on that domain with an increasing number of rooms n connected so that they form a cycle: for each room k, $k \neq n$, there is a locked door k leading to room $k+1$, and an additional, unlocked one between rooms n and 0 (each connection being valid only in the described direction). For each door k there is a key k which is placed in room k, with the exception of key 0 which is placed in room n and the key to the already unlocked door n which doesn't exist. Obviously a good excuse for every n remains the same: If the robot held the key to door 0 in the

[3]This is only possible if that cyclic excuse was already part of the task, but didn't cause a problem because there existed another, after the deletion nonexistent way to the goal.

rooms	sat	opt	rooms	sat	opt
3	0.91s (1)	0.97s (1)	10	19.20s (2)	368.09s (1)
4	1.2s (1)	1.72s (1)	11	57.39s (2)	849.69s (1)
5	1.75s (1)	4.23s (1)	12	72.65s (2)	1175.23s (1)
6	2.19s (2)	10.69s (1)	13	84.45s (2)	—
7	4.24s (2)	27.01s (1)	14	215.05s (2)	—
8	6.03s (2)	65.15s (1)	15	260.39s (2)	—
9	14.22s (2)	158.28s (1)	16	821.82s (2)	—

Table 2: Results for finding excuses on the KEYS domain. We used the same settings as in the experiments for Table 1, except that the weight in the satisficing run was 1. The rows labeled **rooms** give the number of rooms or the size of the cycle minus 1.

initial state, or if that door was unlocked, the task would easily be solvable. The number of necessary regression steps to find that excuse grows with n, though, which is why KEYS is very well suited to test the performance of the finding cyclic excuses part of our implementation.

As can be seen in the results in Table 2, the planning time both scales well with the size of the cycle and is reasonable for practical purposes.

Related Work

We are not aware of any work in the area of AI planning that addresses the problem of explaining why a goal cannot be reached. However, as mentioned already, there is some overlap with abduction (a term introduced by the philosopher Peirce), counterfactual reasoning (Lewis 1973), belief revision (Gärdenfors 1986), and consistency-based diagnosis (Reiter 1987). All these frameworks deal with identifying a set of propositions or beliefs that either lead to inconsistencies or permit to deduce an observation. There are parallels to our notions of acceptable, good and perfect approaches in these fields (Eiter and Gottlob 1995) – for non-cyclic excuses. The main difference to the logic-based frameworks is that in our case there is no propositional or first-order background theory. Instead, we have a set of operators that allows us to transform states. This difference might be an explanation why cyclic excuses are something that appear to be relevant in our context, but have not been considered as interesting in a purely logic-based context.

Conclusion

In this paper we have investigated situations in which a planner-based agent is incompetent to find a solution for a given planning task. We have defined what an *excuse* in such a situation might look like, and what characteristics such an excuse must fulfill to be accounted as *acceptable*, *good* or even *perfect*. Our main theoretical contribution is a thorough formal analysis of the resulting problem along with the description of a concrete method for finding excuses utilizing existing classical planning systems. On the practical side, we have implemented this method resulting in a system that is capable of finding even complicated excuses in reasonable time which is very helpful both for debugging purposes and in a regular setting like planner-based robot control.

As future work, we intend to extend our implementation to more expressive planning formalisms dealing with time and resources.

Acknowledgements

This research was partially supported by DFG as part of the collaborative research center SFB/TR-8 Spatial Cognition Project R7, the German Federal Ministry of Education and Research (BMBF) under grant no. 01IME01-ALU (DESIRE) and by the EU as part of the Integrated Project CogX (FP7-ICT-2xo15181-CogX).

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Comp. Intell.* 11(4):625–655.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *JAAMAS* 19(3):297–331.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *AIJ* 69(1–2):165–204.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Univ. Freiburg, Institut für Informatik, Freiburg, Germany.

Eiter, T., and Gottlob, G. 1995. The complexity of logic-based abduction. *Jour. ACM* 42(1):3–42.

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

Gärdenfors, P. 1986. Belief revision and the Ramsey test for conditionals. *The Philosophical Review* XCV(1):81–93.

Geffner, H. 2000. Functional STRIPS: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Dordrecht, Holland: Kluwer.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS 2009*, 162–169.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26:191–246.

Lewis, D. K. 1973. *Counterfactuals*. Cambridge, MA: Harvard Univ. Press.

Plöger, P.-G.; Pervölz, K.; Mies, C.; Eyerich, P.; Brenner, M.; and Nebel, B. 2008. The DESIRE service robotics initiative. *KI* 4:29–32.

Reiter, R. 1987. A theory of diagnosis from first principles. *AIJ* 32(1):57–95.

Savitch, W. J. 1980. Relations between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences* 4:177–192.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In *ICAPS 2004*, 393–401.