Practical Undoability Checking via Contingent Planning

Jeanette Daum and Alvaro Torralba and Jörg Hoffmann
Saarland University
Saarbrücken, Germany
{lastname}@cs.uni-saarland.de; s9jedaum@stud.uni-saarland.de

Patrik Haslum and Ingo Weber
NICTA
Canberra and Sydney, Australia
{firstname.lastname}@nicta.com.au

Abstract
We consider a general concept of undoability, asking whether a given action can always be undone, no matter which state it is applied to. This generalizes previous concepts of invertibility, and is relevant for search as well as applications. Naïve undoability checking requires to enumerate all states an action is applicable to. Extending and operationalizing prior work in this direction, we introduce a compilation into contingent planning, replacing such enumeration by standard techniques handling large belief states. We furthermore introduce compilations for checking whether one can always get back to an at-least-as-good state, as well as for determining partial undoability, i.e., undoability on a subset of states an action is applicable to. Our experiments on IPC benchmarks and in a cloud management application show that contingent planners are often effective at solving this kind of problem, hence providing a practical means for undoability checking.

Introduction
One essential property of actions is undoability, whether or not their effects can be undone. Informally, an action a is undoable iff whenever a is applied in a state s, there exists a plan for returning to s from the resulting state. Undoability is relevant for a variety of search algorithms, because only non-undoable actions may lead to dead-end states. The special case of undoability in a single step (termed invertibility by Koehler and Hoffmann, 2000; and symmetry by Jonsson et al., 2000), for which syntactic sufficient conditions exist, has been exploited, as has the related concept of reversibility, defined as always being able to return to the initial state (Chen and Giménez 2010). However, no practical support for recognizing undoability (or reversibility) in the general case has been provided.

Undoability is also important in a range of planning applications: For example, Williams and Nayak’s (1997) reactive planner for autonomous spacecraft control had to meet the requirement that non-undoable actions can be used only when repairing failures, not in normal operation. Weber et al. (2012; 2013) argue that undoability checking is important in cloud management, as a tool for administrators dealing with (a complex and ever-growing set of) fixed APIs whose operations may have irreversible side effects depending on context. For example, many operations (adding a slave to a database server, scaling up or down an instance) require stopping and restarting a server, which may overwrite properties not restorable (using the API) later on. Weber et al. employ a PDDL-based undoability checker that naively enumerates the states an action is applicable in. They also propose a simple relevance pruning technique making this enumeration feasible in their domain, yet sacrificing soundness: undoability cannot be concluded with certainty.

However, state enumeration can be avoided by drawing on the power of planning under uncertainty to deal with large belief states. (This was also observed by Eiter et al. (2007; 2008), who proposed, but did not evaluate, a QBF encoding of bounded-length conformant planning for a related problem.) We devise a practical compilation of undoability checking into contingent planning. Given an action a to check, the initial belief state is the set of possible outcome states s’ for all states s that a can be applied in. The goal is to get back, from every s’, to the respective previous state s. Contingent planning allows the undo-plan to observe and branch on “environment variables”, not touched by the action. This is required to decide undoability exactly, as different undo-actions may be needed in different states.

It is sometimes not possible, but also not necessary, to find an undo-plan that returns exactly to s for every state s. Hence, we also present compilations for two useful variations of undoability. First, whether a plan exists to return to a state that is at least as good as s, in terms of goal reachability. With no negative preconditions or goals, a state with more facts true is always better. We name this property rectifiability. Second, if an action is not undoable in every state, we determine partial undoability/rectifiability, by identifying a subset of states on which the action is undoable/rectifiable. Experiments on IPC benchmarks as well as Weber et al.’s cloud management application shows deciding undoability is feasible in many cases, though scalability is problematic in some domains.

We give proof sketches in the text. Full proofs are available in a TR (included as supplementary material).

Background
We base our formalization on the STRIPS framework. STRIPS planning tasks are tuples \( T = (F, A, I, G) \) of facts, action set, initial state \( I \subseteq F \), and goal \( G \subseteq F \). Each \( a \in A \)
Undoability Checking

We consider the problem of checking whether an action $a$ is universally undoable, i.e., undoable for all $s \in S[a]$. Our compilation uses initial state uncertainty to formulate this test compactly. In the compiled planning task, the outcome states $s'$ become the initial belief, and the goal ensures that, for every outcome state $s'$, the contingent plan leads us back to the previous state $s$ corresponding to $s'$. To indicate these roles, we denote the initial belief of the compiled task by $\phi_{s'}$, denote its possible initial states by $s'$, and denote its goal by $G_s$. We refer to plans for the compiled task as undo plans.
Assume the viewpoint of wishing to check whether \( a \) is undoable in \( s \), without actually knowing what \( s \) is beyond the fact that \( a \) is applicable in \( s \), i.e., \( s \in S[a] \). What do we know about the outcome state \( s' := s[a] \)? We know that \( add(a) \) and \( pre(a) \backslash \ delete(a) \) must be true, and we know that \( del(a) \) must be false. What we do not know are the values of the environment variables \( F^E(a) := F \backslash (pre(a) \cup add(a) \backslash delete(a)) \) not touched by \( a \) at all. As for \( s \) itself, we know that \( pre(a) \) must be true, but we do not know the values of the environment variables, nor those of the overwritten variables \( F^O(a) := add(a) \cup (delete(a) \backslash pre(a)) \) which are set by \( a \)'s effects but are not constrained by \( a \)'s precondition.

We handle overwritten variables simply by enumeration, i.e., we create one compiled task for every value assignment to \( F^O(a) \). While the number of such assignments is exponential, the exponent \( |F^O(a)| \) can be expected to be small in practice.\(^2\) In contrast, as STRIPS actions typically touch only a small fraction of the fact set \( F \), the number of environment variables \( |F^E(a)| \) will typically scale linearly with the size of the input. So these we need to handle in a compact way. Our compilation does so by maintaining, for each \( p \in F^E(a) \), additional variables \( WasTP^p \), \( WasFP^p \), and \( Ok^p \). These reflect whether \( p \) was true or false in the pre-application state \( s \) and whether in the current state of our undo plan we know that \( p \) has that same value again, respectively. To ascertain this behavior, we augment the original task’s actions as follows:

**Definition 2** Given a STRIPS action \( a \), by \( a^U \) we denote the action identical to \( a \) with conditional effects \( E(a^U) = \{
\begin{array}{ll}
(\{\text{WasTP}^p\},\{\text{Ok}^p\},\emptyset), (\{\text{WasFP}^p\},\emptyset,\{\text{Ok}^p\}) & | p \in \text{add}(a) \cup F^E(a) \\
(\{\text{WasFP}^p\},\{\text{Ok}^p\},\emptyset,\{\text{Ok}^p\}) & | p \in \text{del}(a) \cap F^E(a)
\end{array}\}
\)

If an action \( a \) makes \( p \) true, then \( p \) is deemed to be Ok if it was true in \( s' \), and not-Ok if it was false in \( s' \). The case where \( a \) makes \( p \) false is handled symmetrically.

We need a last simple notation, associating an individual fact \( p \) with a straightforward observation action \( ap^p \):

**Definition 3** The observer of a fact \( p \) is the observation action \( ap^p \) where \( pre(ap^p) = \emptyset \) and \( obs(ap^p) = p \).

Such observation actions will enable the contingent undo plan to branch on the values of environment variables.

In what follows, we represent value assignments to \( F^O(a) \) as subsets \( O \subseteq F^O(a) \), i.e., the subset of facts assigned to true. Slightly abusing notation, we identify fact sets with fact conjuctions, and assume negated goals as a syntactic feature of contingent planning tasks. Negated goals can be compiled away with standard techniques (Gazend and Knoblock 1997); many tools (e.g., Contingent-FF) support them at PDDL level. Our compilation can then be stated as follows:

**Definition 4** Let \( \Pi = (F, A, I, G) \) be a STRIPS planning task, \( a \in A \) an action, and \( O \subseteq F^O(a) \). The undoability-compilation of \( a \) with respect to \( O \) is the contingent planning task \( \Pi^U[a, O] := (F_{\Pi^U}, A_{\Pi^U}, \phi_{\Pi^U}, G_{\Pi^U}) \) as follows:

\(^{(i)}\) \( F_{\Pi^U} = F \cup \{ WasTP^p, WasFP^p, Ok^p | p \in F^E(a) \} \);
\(^{(ii)}\) \( A_{\Pi^U} = A_r \cup A_o \) where \( A_r = \{ a^U | a \in A \} \) and \( A_o = \{ ap^p | p \in F^E(a) \} \);
\(^{(iii)}\) \( \phi_{\Pi^U} = add(a) \land (pre(a) \backslash del(a)) \land \{ \neg p | p \in del(a) \} \land \bigwedge_{p \in F^E(a)} Ok^p \land \bigwedge_{p \in F^E(a)} (p \leftrightarrow WasTP^p) \land (p \leftrightarrow WasFP^p) \); and
\(^{(iv)}\) \( G_{\Pi^U} = pre(a) \cup \{ Ok^p | p \in F^E(a) \} \cup O \cup \{ \neg p | p \in F^O(a) \backslash O \} \).

Items (i) and (ii) should be clear given the discussion above. To understand the specification of \( \phi_{\Pi^U} \), recall our knowledge about the outcome state \( s' : add(a) \) and \( pre(a) \backslash del(a) \) must be true, and \( del(a) \) must be false. The value of the environment variables \( F^E(a) \) is unknown; \( \phi_{\Pi^U} \) says that, at the start of the undo plan, every \( p \in F^E(a) \) is Ok, and it sets \( WasTP^p \) and \( WasFP^p \) according to the value of \( p \) in every \( s' |\sim \phi_{\Pi^U} \). The undo-goal \( G_{\Pi^U} \), finally, constrains \( pre(a) \) and the environment variables as expected. Beyond that, we need to ensure that we can get back to the desired setting \( O \) of the overwritten variables in \( s \). Note that this is the only point in the compilation where \( O \) plays a role (in \( s' \), these values are known simply because they are set by \( a \)'s effects).

**Theorem 1** Let \( \Pi = (F, A, I, G) \) be a STRIPS planning task. An action \( a \in A \) is universally undoable if and only if, for all \( O \subseteq F^O(a) \), \( \Pi^U[a, O] \) is solvable.

**Proof Sketch:** \( \Rightarrow \): Say that \( a \) is universally undoable, and let \( O \subseteq F^O(a) \). A plan \( T \) for \( \Pi[a, O] \) can be constructed by observing the values of all environment variables, and attaching to each leaf node, which must correspond to a unique state \( s' |\sim \phi_{\Pi^U} \), a corresponding sequence \( \delta_{s',s'} \) with \( s' |\sim \delta_{s',s} = s' \), where \( s \) is chosen such that \( s' = s[a] \). By construction, \( \delta_{s',s} \) results in a goal belief state of \( \Pi[a, O] \).

\( \Leftarrow \): Say that \( s \in S[a] \) and \( s' = s[a] \). Let \( O := F^O(a) \cap s \) be the assignment to \( a \)'s overwritten variables in \( s \). Let action tree \( T \) be a plan for \( \Pi[a, O] \). Executing \( T \) on \( s' \), let \( (a_1^U \ldots a_n^U) \) be the resulting sequence of applicable regular actions from \( A_{\Pi^U} \) which achieves \( G_{\Pi^U} \). Let \( \delta_{s',s} := (a_1^U \ldots a_n^U) \). By construction, \( s' |\sim \delta_{s',s} = s \).

For undoability checking (as opposed to universal-undoability checking as in Theorem 1), our compilation can be strengthened with invariants, i.e., formulas that hold in all reachable states. Such invariants can be found by several known methods (e.g., Blum and Furst 1997; Fox and Long 1998; Gerevini and Schubert 2000; Scholz 2000; Rintanen 2000; Helmert 2009). If \( \psi \) is an invariant formula, we can replace \( \phi_{\Pi^U} \) with \( \phi_{\Pi^U} \land \psi \) in \( \Pi^U[a, O] \). Furthermore, if \( pre(a) \land \psi \) entails a specific value for some \( p \in F^O(a) \), we can remove \( p \) from \( F^O(a) \) as its value is known. In both cases, solvability of all \( \Pi^U[a, O] \) remains a sufficient criterion for undoability of \( a \). This is important because actions are often not universally undoable, but can be proven are undoable in reachable states thanks to invariants. For example, in Barman, “leave(h c)” is undoable by “grasp(h c)”, but only in reachable states. In the unreachable states \( s \) where “[holding h c)” and “[ontable c)” are both true, “leave(h c)” cannot be undone, because “grasp(h c)” deletes “[ontable c)”. In other words, we cannot get back to the previous inconsistent state. Restricting \( \phi_{\Pi^U} \) with the mutex invariant

\(^2\)An alternative is to introduce observation actions for the overwritten variables; we will get back to this below.
"\neg(holding ~ h \land \neg(ontable ~ c))", we get rid of the inconsistent states, resulting in solvable tasks $\Pi^U[a, O]$ and thus a proof of undoability.

Using contingent – rather than conformant – planning allows the undo plans to branch on environment variables. An example of why this is necessary is the “switch-off” action in IPC Satellite. To undo it, we need to apply “switch-on”, which deletes calibration, so we need to recalibrate, turning the satellite from its current direction to a calibration target. The undo plan hence consists of a case distinction over the “pointing(...)” environment variables, encoding the direction of the satellite.3

In general, branching over environment variables is needed when actions have (a) different enabled preconditions or (b) different conflicts with the need to preserve environment variable values. To illustrate (b), say we wish to undo $a$ which deletes $p$, $a_1$ adds $p$ and environment variable $q$, and $a_2$ adds $p$ but deletes $q$ (and there are no other actions). Then $a_1$ can only be used if $q$ was true beforehand, while $a_2$ can only be used if $q$ was false beforehand.

On the other hand, branching over environment variables is not needed if neither of (a) or (b) apply, i.e., if the undo plan neither relies on the values of environment variables nor affects them. So, in many cases, the contingent undo plan makes do with very little or no branching. This is different for the overwritten variables. In principle, one could handle these in a similar manner, encoding their prior values as unknown facts $p$ in $\phi_s$, and including observation actions $a^p$ for them (unless, that is, we can infer their values from $\text{pre}~(a)$ and invariants). This would tackle the entire undoability check with a single call of contingent planning. However, in difference to $p \in F^E(a)$, for $p \in F^O(a)$ we can not assume $Ok^p$ in $\phi_s$: the values of overwritten variables are affected by the action, and may have been different in the preceding state $s$. Hence the contingent plan would have to establish $Ok^p$ for all $p \in F^O(a)$, which would necessarily involve including branches enumerating all value assignments to $F^O(a)$. It is more effective to do this enumeration outside the planner.

Anyway, as we shall see in the next section, once we modify undoability to rectifiability, i.e. require only that we can always get back to an at-least-as-good state, it suffices to consider a single assignment to the overwritten variables: The one where they are all assigned to true.

From Undoability to Rectifiability

We straightforwardly modify Definition 1:

Definition 5 Let $\Pi = (F, A, I, G)$ be a STRIPS planning task, and $a \in A$ an action. For $s \in S[a]$, $a$ is rectifiable in $s$ if there exists an action sequence $\delta_{s,s'}$ such that $s[a][\delta_{s,s'}] \supseteq s$. We say that $a$ is rectifiable if it is rectifiable for all reachable states $s \in S[a]$, and we say that $a$ is universally rectifiable if it is rectifiable for all $s \in S[a]$.

Similar to the use of invertibility as a simple syntactic special case of undoability, prior work (Hoffmann 2005) has identified a special case of rectifiability. An action $a$ is at-least-invertible if there is an action $\pi \in A$ such that (1) $\text{pre}(\pi) \subseteq \text{pre}(a) \cup \text{add}(a) \setminus \text{del}(a)$; (2) $\text{add}(\pi) \supseteq \text{del}(a)$; (3) each fact in $\text{del}(\pi)$ is mutex with at least one fact in $\text{pre}(a)$. In this situation, (1) $\pi$ is always applicable behind $a$, (2) $a$ re-achieves everything deleted by $a$, and (3) does not delete anything true prior to the application of $a$. Every invertible action is also at-least-invertible, so this syntactic special case generalizes the previous one. For example, the “rewind-movie” action $a$ in the Movie domain is not undoable (and hence, in particular, not invertible) because it adds “movie-rewound”, which is not deleted by any action. However, the only fact deleted by “rewind-movie” is “counter-at-zero”. This can be re-achieved by the “reset-counter” action, which has empty preconditions and deletes and thus qualifies for the role of $\pi$ as specified above.

It turns out that a simplified version of our previous compilation is suitable for checking rectifiability:

Definition 6 Given a STRIPS action $a$, by $a^R$ we denote the action identical to $a$ but with conditional effects $E(a^R) = \{((\{\text{Was}~T^p\}, \{Ok^p\}, 0) \mid p \in \text{add}(a) \cap F^E(a)) \cup \{((\{\{\text{Was}~T^p\}, \{Ok^p\}\} \mid p \in \text{del}(a) \cap F^E(a))\}.

Let $\Pi = (F, A, I, G)$ be a STRIPS planning task, and $a \in A$. The rectifiability-compilation of $a$ is the contingent planning task $\Pi^R[a] := (F_{a,s}'', A_{a,s}'', \phi_{a,s}'', G_s)$ as follows:

(i) $F_{a,s}'' = F \cup \{\{\text{Was}~T^p\}, \{Ok^p\}, 0\mid p \in F^E(a)\}$;
(ii) $A_{a,s}'' = A' \cup A_s$, where $A_s = \{a^R \mid a \in A\}$ and $A'_s = \{a^p \mid p \in F^E(a)\}$;
(iii) $\phi_{a,s}'' = \text{add}(a) \cup \text{del}(a) \cup \text{\neg p \mid p \in \text{del}(a)} \cup \bigwedge_{p \in F^E(a)} \text{Ok}^p \cup \bigwedge_{p \in F^E(a)} \text{Ok}^p \cup \bigwedge_{p \in F^E(a)} \text{Ok}^p$; and
(iv) $G_s = \text{pre}(a) \cup \{\text{Ok}^p \mid p \in F^E(a)\} \cup F^O(a)$.

Relative to the previous augmented actions (Definition 2), we do not have to maintain the $\text{Was}~T^p$ flags, because we don’t care if an environment variable was previously false and has been made true by the undo plan. Relative to the previous compiled planning task (Definition 4), we drop the $\text{Was}~T^p$ flags from $\phi_{a,s}'$, and we no longer need the input argument $O$ enumerating assignments to the overwritten variables $F^O(a)$. Instead, we just constrain these variables to be true in the goal. This suffices to get the desired equivalence:

Theorem 2 Let $\Pi = (F, A, I, G)$ be a STRIPS planning task. An action $a \in A$ is universally rectifiable if and only if $\Pi^R[a]$ is solvable.

Proof Sketch: $\Rightarrow$: If $a$ is universally rectifiable, to construct a plan for $\Pi^R[a]$ we observe the values of all environment variables as in Theorem 1, and attach to each leaf node corresponding to state $s'$ a sequence $\delta_{s,s'}$ where $s$ is chosen such that $s' = s[a]$ and $F^O(a) \subseteq s$. The latter property can be ensured WLOG because $s = s[a]$ is not affected by the values of the overwritten variables in $s$.

$\Leftarrow$: Say that $s \in S[a]$ and $s' = s[a]$, that $T$ is a plan for $\Pi^R[a]$, and that executing $T$ on $s'$ yields the sequence $\langle a_1, \ldots, a_n \rangle$ of regular actions. Then $\delta_{s,s'} := \langle a_1, \ldots, a_n \rangle$ satisfies $s'[\delta_{s,s'}] \supseteq s$ as desired because, by construction,
all environment variables true in $s$, as well as all overwritten variables, are true in $s'[\hat{a}_{n,s'}]$. □

Intuitively, setting all overwritten variables to true is enough because, there being no need to rectify a false variable, this is the worst thing we may need to rectify. Given this, one may be tempted to think that a similar “worst-case” handling is possible for the environment variables. But this is not so because, for environment variables, being true is “bad” because the undo plan must ensure they are true again, while being false is “bad” because it enables less undo-plan actions (the latter is not so for overwritten variables whose values in $s'$ are fixed anyhow). To illustrate, say we wish to undo $a$ which deletes $p$, $a_1$ adds $p$ and has the environment-variable precondition $q$, and $a_2$ adds $p$ but deletes $q$ (and there are no other actions). Then $a_1$ can only be used if $q$ was true beforehand, while $a_2$ can only be used if $q$ was false beforehand, so, like for undoability, different environment variable values may necessitate different actions.

**Partial Undoability/Rectifiability**

If an action is not undoable, or not rectifiable, on all states, naturally it is of interest to ask for subsets of states that work:

**Definition 7** Let $\Pi = (F, A, I, G)$ be a STRIPS planning task, $a \in A$, and $S \subseteq S[a]$. We say that $a$ is undoable (rectifiable) in $S$ if $a$ is undoable (rectifiable) in all $s \in S$.

In what follows, to avoid clumsy language we will mostly talk about undoability only, the understanding being that all claims apply also to rectifiability exactly as stated.

As a major challenge beyond our previous analysis techniques, analyzing the partial undoability of an action $a$ necessitates to find a state set $S$ on which $a$ is undoable. How to do that, and how to make $S$ as large as possible?

One possibility lies in different contingent planning algorithms, that do not attempt to find a plan solving all possible initial states, but instead try to solve many such states. From the existing literature, the most suitable tool support in this direction comes from online contingent planners (Albore et al. 2009; Shani and Brafman 2011; Brafman and Shani 2012a; 2012b; Malia et al. 2014). The algorithms underlying these tools are designed to interleave planning and execution. To simulate execution, the outcome of observation actions is chosen based on some preselected initial state. For our purposes, this results in a machinery choosing observation outcomes to narrow down the search space in the (offline) analysis of partial undoability. Given that observation actions are chosen in a manner trying to minimize their use while maximizing progress to the goal, this provides some support for obtaining large sets $S$ of solved initial states.

It turns out one can also compile the problem into standard contingent planning, resulting in plans that maximize the solved initial states, in a particular sense. The compilation per se is simple: We introduce a give-up action $a_{\text{GiveUp}}$ with empty precondition, effect $G$, and an action cost $C$. This allows to achieve the goal anytime, at cost $C$. Given a contingent task $\Pi$, we denote this compiled task by $\Pi|_C$.

For this simple compilation to behave as intended, we need to suitably arrange (the value of $C$ and) the optimization metric i.e., the “cost” of AND nodes (observation actions) in the plan. The two most commonly used definitions are sum aggregation, minimizing the overall size of the plan tree, where the cost of an observation action node $N$ for action $a \in A_s$ is $c(N) := c(a) + \sum_{i=1}^{n} c(N_i)$ with $N_i$ being $N$’s children; and max aggregation, minimizing the maximum-depth of the plan tree, where $c(N) := c(a) + \max_{i=1}^{n} c(N_i)$. However, as we shall see next, our compilation here is most sensible under a third variant: average aggregation, where $c(N) := c(a) + \frac{1}{\pi} \sum_{i=1}^{n} c(N_i)$.

For input tasks $\Pi$ that have a full contingent plan, trivially, for all three aggregation methods, with sufficiently large $C$ (larger than the cost of at least one plan) the give-up action is not used, and a full contingent plan solving all initial states is returned. However, for partial undoability checking, we are interested only in the case where a full contingent plan (a full undo-plan) does not exist. Max and sum aggregation then are useless because $a_{\text{GiveUp}}$ is an optimal plan for $\Pi|_C$: every plan must contain $a_{\text{GiveUp}}$, and adding other actions can only increase plan cost. In contrast, for average aggregation, we can set $C$ so that the contingent plan gives up on a belief $b$ only if $b$ does not even have a weak plan (Cimatti et al. 2003); if there does not exist any sequence of observations and actions leading from $b$ to the goal. Referring to beliefs that have a weak plan as partially solvable, we have:

**Theorem 3** Let $\Pi = (F, A, I, G)$ be a contingent planning task. Let $B := 2^{2^{2m}}$ be the size of the belief space, and let $C := (2^B - 1) * B + 1$. Let $T_0$ be a plan for $\Pi|_C$ that is optimal under average aggregation. Then $T_0$ uses $a_{\text{GiveUp}}$ only on beliefs that are not partially solvable.

**Proof Sketch:** Say that $b$ is partially solvable. Let $\pi$ be a weak plan for $b$. Consider the action tree $T$ constructed by following $\pi$ and applying $a_{\text{GiveUp}}$ to each observation-action outcome not contained in $\pi$. As the number of observation actions on $T$, and the number of regular actions in between observations, is bounded by $B$, $2^{B-1} (B + C)$ is an upper bound on the average-aggregation cost of $T$, and hence on that of $T_0$. So for any setting of $C$ s.t. $C > 2^{B-1} (B + C)$, $T_0$ cannot use $a_{\text{GiveUp}}$ on $b$. The claim follows as $C > (2^B - 1) * B$. □

Given this, to test partial undoability/rectifiability, one can use the give-up compilation on top of our previous compilations: $\Pi^U[a, O]|_C$ respectively $\Pi^R[a]|_C$. As average aggregation is not supported by standard contingent planning tools, we implemented it in Contingent-FF (CFF) (Hoffmann and Brafman 2005). CFF does not support action costs, so our implementation is native inside its AO* search algorithm. Also, as CFF’s heuristic may return $\infty$ on partially solvable beliefs, we added a new heuristic (for partial undoability only), which assumes all unknown facts are true and then runs the standard $h^P$ heuristic.

---

4We remark that, for $\Pi^U[a, O]$ and $\Pi^R[a]$ where environment variables can always be observed, the much smaller setting $C := (2^{h^P(a)} - 1) * 2^{h^P} + 1$ suffices for the claim of Theorem 3.
Experiments

We implemented our compilations at lifted-PDDL level, i.e., while each test pertains to a ground action \( a \), the action sets of the compiled contingent planning tasks \( \Pi^R[a] \) and \( \Pi^R[a] \) are encoded as parameterized action schemas. This is important in practice, as otherwise the compiled task can become so large as to cause difficulties for planner parsers. We use binary mutexes and invariants found by Scholz’ (2000) algorithm\(^7\), and the FD translator (Helmert 2009) to generate the ground actions to be tested.

We focus in what follows on rectifiability testing, due to space restrictions and as rectifiability is arguably the more relevant property in practice. For the same reason, throughout we enhance our rectifiability tests with reachability information, not examining the behavior without it.

The reachability information we use are pairwise fact mutexes and exactly-1 invariants as detected by pddlcat. These are added to the initial belief \( \phi_s \) as previously described, which for mutexes \( \neg p \lor \neg q \) where \( p \in \text{pre}(a) \) is equivalent to restricting \( a \) with the implicit precondition \( \neg q \). Note that in this case, \( q \notin F^O(a) \) even if \( q \in \text{add}(a) \). We also employ a simple form of admissible relevance pruning. Starting with \( R_A := \emptyset \) and \( R_F := \text{del}(a) \) where \( a \) is the action to be tested, we iteratively include into \( R_A \) those actions \( a' \) whose add list intersects \( R_F \), and include into \( R_F \) those facts appearing in \( \text{pre}(a') \cup \text{del}(a') \). Upon reaching a fixed point, we know that \( R_F \) contains all facts that an undo plan might need to make true, and that \( R_A \) contains all actions that may be needed for that purpose. We can thus, without affecting the outcome of the test, remove from \( \Pi^R[a] \) those actions and facts not in \( R_A \) respectively \( R_F \), as well as any observation actions pertaining to facts outside \( R_F \).

All experiments were run on a cluster of Intel E5-2660 machines running at 2.20 GHz, with time/memory cut-offs for each call of contingent planning set to 10 minutes/4 GB.

Contingent Planners

For full (as opposed to partial) rectifiability testing, we ran CFF and the offline version of CLG (Albore et al. 2009). PO-PRP (Muise et al. 2014) is not applicable because our compiled tasks are not “simple” (unknown literals occur as effect conditions). The compilation approach of CLG, on the other hand, is perfectly suitable in principle as the contingent width of our compiled tasks is 1. Unfortunately, the compiled tasks are quite large and, despite the width 1, CLG’s translation step often is prohibitive. Furthermore, there is a bug in CLG’s implementation which sometimes leads to invalid undo plans.\(^6\) So the main planner in our experiments here is CFF, which turned out to almost consistently outperform CLG. We include a brief summary of CLG’s performance below. We run CFF without helpful actions pruning so as to allow proving unsolvability (non-rectifiability).

For partial rectifiability testing, we run CFF with the compiled tasks \( \Pi^R[a]_{\perp C} \); as the exclusive focus here is finding plans, rather than proving their absence, we do use helpful actions pruning. Regarding the alternative approach, online contingent planning on the tasks \( \Pi^R[a] \), we did not experiment with CLG because it may return invalid undo plans and we have no means for plan checking (which is itself a hard problem in contingent planning). We did run experiments with SDR (Brafman and Shani 2012b), which is more recent anyhow and represents the state of the art in this area.

IPC Benchmarks

We run all STRIPS domains from IPC’98 to IPC’14. For domains that occur several times, we use the most recent instance suite; for domains that have both an optimal and a satisfying instance suite, we use the optimal one (due to scaling, see below). As many benchmarks have large numbers of ground actions, rather than testing all of these, for every ground action \( a \) we first test whether \( a \) is at-least-invertible, and if so (record and) discard \( a \). From the remaining ground actions, we randomly select 50 per action schema. It makes sense to organize these experiments per schema, as ground actions of different schemas typically exhibit characteristically different behavior. We omit pre-grounded domains (no action schemas), and we select only four instances from each test suite, namely the smallest-index one, the largest-index one, and two instances at equal spacing (with respect to the IPC indexing) in between. Overall, this still results in 48800 compiled contingent planning tasks. Consider Table 1.

Of the 35 domains, in the at-least-invertible criterion applies to all actions (Blocksworld, Driverlog, Elevators, Gripper, Logistics, Miconic, Movie, Transport, VisitAll). In Tidybot, the compiled tasks are too large for CFF to parse. In 4 domains, nothing or not much is gained over the invertibility test (Mprime, the Pipesworlds, Tetris). In 4 domains, we find undo plans (Miconic, Satellite, Scanalyzer, Zenotravel). In the remaining 17 domains, many actions are proved non-rectifiable. In 6 domains this happens reliably (Childsnake, Floortile, Hiking, PegSol, Sokoban, TPP); in the other 11 domains, the ability to prove non-rectifiability varies depending on action schema and instance size.

The successful “YES” cases are usually proved quickly. If an undo plan is found, CFF does so in < 1 second. The only exception is Satellite, where CFF finds complex undo plans for “switch-off” (20 regular actions, 6 observation actions, tree depth 10) respectively “switch-on” (26 regular, 6 observation, tree depth 10), in 261.2 respectively 265.5 seconds.

The vast majority of successful “NO” cases are proved either (a) by CFF’s pre-processing or (b) CFF’s heuristic function returning \( \infty \) on the initial state. (a) happens, on our encodings, iff the goal is not reachable in a relaxed planning graph even when assuming that all environment variables are true. (b) happens iff there exists a value assignment to the environment variables for which the goals are not reachable in a relaxed planning graph. Only few “NO” cases are proved by (c) the actual search. This is unsurprising as the corresponding IPC instances already have hundreds or thousands of facts and actions, so the contingent compilation, yet larger than this, cannot be proved unsolvable by belief-space search. Success in the “NO” case is, therefore, largely determined by whether or not (a) or (b) apply. Concretely, 52%
of the successful “NO” cases are of category (a), proved in mean/maximum time 0.1/0.9 seconds; 47% of cases are of category (b), proved in mean/maximum time 6.1/32.8 seconds; and 1% of cases are of category (c), proved in mean/maximum time 31.2/203 seconds.

The performance of CLG is almost consistently worse than that of CFF. A major advantage of CFF is its ability to handle large non-undoable instances, thanks to (a) and (b) above. For such instances, CLG’s compilation either explodes in size and runs out of memory, or it does not carry over to classical planning the structure allowing CFF to easily detect unsolvability. The single case where CFF successfully generates more ground actions than CFF is Woodworking “do-grind”, where CFF proves 29% of the ground actions non-undoable (vs. CFF’s 9%).

Analyzing partial rectifiability using CFF and our give-up compilation turns out to be hard because, prior to giving up, search must explore all plan trees whose average cost is below the give-up cost $C$. With large values of $C$, this is infeasible in our experiments. Setting $C := 3$, we find partial undo-plans for Depots “lift”, Grid “move”, Rovers “take-image”, Thoughtful “turn-deck-a” and Woodworking “do-plane”. Each of these undo plans works “in half of the cases”: the undo plan applies one sensing action and solves one case while giving up on the other case. The single exception is Woodworking, where the undo plans apply two sensing actions and solve three of the four possible cases.

The online contingent planner SDR finds partial undo plans in Depots, Grid, Parking, and Satellite. The only improvement over CFF is in Parking, for some instances of “move-car-to-car”. For the remaining 19 domains of interest (those with non-rectifiable actions), in 7 domains we could not get SDR to run, in 8 domains SDR always runs out of time or memory, and in 4 domains SDR always gets stuck in a dead end, i.e., the preselected initial state is unsolvable and hence no undo plan is found.

Cloud Management

Weber et al.’s (2012; 2013) cloud management application is interesting from our perspective as a realistic use case, established and investigated in practically oriented prior work. For administrators, being aware of action undoability is of the essence, as it is most embarrassing to cause a lasting outage by incorrect API usage. Support for analyzing undoability is valuable as the set of APIs to deal with is complex, grows continually, and involves many operations with irreversible (side) effects depending on context. Furthermore, the API is the only method available to manipulate the resources – although Web frontends and other tools exist, they

Table 1: Rectifiability checking results with CFF. We omit action schemas where all ground actions are at-least-invertible, and domains where this applies to all schemas. YES: proved rectifiable, NO: proved non-rectifiable. %I: global fraction of at-least-invertible actions. As fractions of the actions selected per schema, across IPC instances: %Y: proved YES, %PN: proved NO by pre-process, %HN: proved NO by initial-state heuristic value, %SN: proved NO by search, %S: successfully analyzed (proved YES or NO). #A: average number of actions in undo plan. C/CA: number of IPC instances (out of the 4 selected per domain) where some/all actions are successfully analyzed.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Action schema name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YES %F</td>
</tr>
<tr>
<td>Barman</td>
<td>(reveal</td>
</tr>
<tr>
<td>Marlinspike</td>
<td>make</td>
</tr>
<tr>
<td>Depots</td>
<td>lift</td>
</tr>
<tr>
<td>Flooring</td>
<td>paint-down</td>
</tr>
<tr>
<td>Fireclay</td>
<td><em>home</em></td>
</tr>
<tr>
<td>Fireclay</td>
<td>move</td>
</tr>
<tr>
<td>Fireclay</td>
<td>send</td>
</tr>
<tr>
<td>Gold</td>
<td>begin</td>
</tr>
<tr>
<td>GED</td>
<td>continue</td>
</tr>
<tr>
<td>GED</td>
<td>continue-inverse</td>
</tr>
<tr>
<td>GED</td>
<td>continue-inverse</td>
</tr>
<tr>
<td>Grid</td>
<td>move</td>
</tr>
<tr>
<td>Grid</td>
<td>unlock</td>
</tr>
<tr>
<td>Hiking</td>
<td>walk-together</td>
</tr>
<tr>
<td>Mpryme</td>
<td>drive</td>
</tr>
<tr>
<td>Mpryme</td>
<td>drink</td>
</tr>
<tr>
<td>Mpryme</td>
<td>feast</td>
</tr>
<tr>
<td>NoMystery</td>
<td>drive</td>
</tr>
<tr>
<td>Parking</td>
<td>move</td>
</tr>
<tr>
<td>Parking</td>
<td>move</td>
</tr>
<tr>
<td>Parking</td>
<td>move</td>
</tr>
<tr>
<td>PipesTank</td>
<td>push-end</td>
</tr>
<tr>
<td>PipesTank</td>
<td>push-end</td>
</tr>
<tr>
<td>PipesTank</td>
<td>push-end</td>
</tr>
<tr>
<td>PipesTank</td>
<td>push-end</td>
</tr>
<tr>
<td>Rovers</td>
<td>drop</td>
</tr>
<tr>
<td>Rovers</td>
<td>sample</td>
</tr>
<tr>
<td>Rovers</td>
<td>sample</td>
</tr>
<tr>
<td>Satellite</td>
<td>switch-off</td>
</tr>
<tr>
<td>Satellite</td>
<td>switch-on</td>
</tr>
<tr>
<td>Scana</td>
<td>lyzer</td>
</tr>
<tr>
<td>Scana</td>
<td>lyzer</td>
</tr>
<tr>
<td>Sokoban</td>
<td>push-to</td>
</tr>
<tr>
<td>Sokoban</td>
<td>push-to</td>
</tr>
<tr>
<td>Sokoban</td>
<td>push-to</td>
</tr>
<tr>
<td>Sokoban</td>
<td>push-to</td>
</tr>
<tr>
<td>Sokoban</td>
<td>push-to</td>
</tr>
</tbody>
</table>
offer the same API operations through other channels.

Weber et al. devise an encoding of 35 API operations, from Amazon Web Services (AWS) Elastic Compute Cloud (EC2), into a PDDL domain file. The APIs, each encoded as an action schema, manipulate cloud resources such as virtual machines (VM) and virtual harddisks (VHD). Action effects concern resource status (e.g., start VM, delete VHD), relationships (e.g., connect VM to VHD), or attributes (change VM from small to large). Preconditions capture requirements, e.g., a VM’s size can only be changed if it is stopped.

For the undoability tests, Weber et al. employ a classical planner. Towards that end, they first create a set of objects, essentially just one representing each type (the underlying assumption being that, within each type, objects behave symmetrically). To test an action, they select a set of actions $U_a$ whose adds cover $\text{add}(a)$ and whose deletes cover $\text{del}(a)$. They deem the facts $F_a$ touched by $U_a$ as relevant, fix all other facts (environment variables) to be false, and create one classical planning task for every pre/post state pair $(s, s')$ over $F_a$ where $\text{pre}(a) \subseteq s$. They consider $a$ to be undoable iff all these planning tasks are solvable.

In brief, Weber et al.’s test is naïvely enumerative, but uses a simple pruning technique to consider only a subset $S \subseteq S(a)$ of states $a$ is applicable to. The pruning technique is, obviously, not sound, in that $S$ may overlook a state $a$ is not undoable on. So the test may report $a$ to be undoable even if that is not so. Note furthermore that the test is agnostic to reachability information, and may report $a$ to be not-undoable due to unreachable pre/post states. One example where this happens is actions of the form “start-instance”, whose precondition “(instanceStopped ?inst)” implies that no IP is associated with ?inst. Without this implicit precondition, the action is not undoable. Weber et al. propose that the API provider (or the system admin) interacts with the undoability checker to add implicit preconditions. Consequently, they have two PDDL domain files, $D_W$ with the implicit preconditions, and $D_{W/O}$ without them.

Weber et al.’s tool is quite effective performance-wise. Using FF (Hoffmann and Nebel 2001), the total planning time across all $a$ is 3 seconds in each of $D_{W/O}$ and $D_W$. On $D_W$, the tool correctly classifies all actions as undoable, except for “create-autoscaling-group” which is correctly classified as non-undoable. On $D_{W/O}$, the actions with missing implicit preconditions are incorrectly classified as being non-undoable. So, in the domain the tool was built for, it does not suffer from its lack of soundness, but does suffer from its lack of reachability information. Using our technology, the same tests can be performed in little more runtime, with soundness guarantee, and automatically finding the necessary reachability information.

Weber et al.’s PDDL contains some universally quantified preconditions, which we transformed to STRIPS in the usual manner (Gazen and Knoblock 1997). On these STRIPS versions of $D_{W/O}$ and $D_W$, Weber et al.’s tool yields the same analysis results in slightly worse runtime. We created a canonical initial state instantiating the objects with their (obvious) initial properties. Running CFF on the contingent planning tasks $\Pi^U[a]$ checking undoability takes 15.3 seconds total time for $D_{W/O}$ and 15.7 seconds for $D_W$, classifying all actions correctly. In particular, the actions with missing implicit preconditions in $D_{W/O}$ are determined to be undoable. Running CFF on the tasks $\Pi^R[a]$ to check rectifiability takes 17.3 (16.1) seconds for $D_{W/O}$ ($D_W$), and determines that “create-autoscaling-group” is (not undoable but) rectifiable. The trivial sound method, naïve enumeration of states in the projection onto the relevant facts $R_F$, would be feasible here, yet would incur substantial overhead: the average (maximum) size of $R_F$ is 9.8 (22).

Related Work

The best-known prior work on undoability detection, in the AI planning community, pertains to the simple sufficient criteria, invertible and at-least-invertible actions (Koehler and Hoffmann 2000; Hoffmann 2005), that we already discussed. Hoffmann (2005) introduces the additional notion of actions whose adds are static and that delete only their own preconditions. If every action either satisfies this criterion, or is at-least-invertible, then the state space (may contain non-rectifiable actions but) cannot contain dead end states.

The only prior work on general undoability checking is that by Eiter et al. (2007; 2008). They propose an encoding into conformant planning under uncertainty. They did not implement this approach though, and it is quite different in the specifics. They propose to create a library of “reverse plan items (RPI),” action sequences $\overrightarrow{a}$ along with corresponding reverse sequences $\overleftarrow{a}$ that will always undo $\overrightarrow{a}$. Once created, the library can be used during plan execution, to identify undo plans if needed. To find the RPIs, Eiter et al. design a fixed-plan-length QBF encoding of a conformant problem guaranteeing that, for some index $i$ along the plan, the prefix $\overrightarrow{a}$ up to $i$ will be undone by the postfix $\overleftarrow{a}$ behind $i$. This forces the conformant planner to find “cycles”, and thus valid RPIs. Towards partial undoability, Eiter et al. consider “conditional” RPIs, and a modified QBF encoding allowing $\overrightarrow{a}$ to mark states as “don’t care”, freeing $\overleftarrow{a}$ from the obligation to undo them.

For our purposes, i.e., deciding whether a specific action $a$ is undoable, Eiter et al.’s approach could be adapted by forcing the use of $a$ in the first time step of their QBF encoding, allowing us to look for reverse sequences $\overrightarrow{a}$ for $a$. However, the length of $\overrightarrow{a}$ is fixed as part of the encoding, and $\overrightarrow{a}$ is a conformant plan so will fail in those cases where different actions must be taken depending on the values of the environment variables.

Conclusion

Undoability checking can be encoded into contingent planning. In practice, perhaps not surprisingly, the problem can often be solved, even proved unsolvable, with reasonable effort, due to the particular nature of “typical” actions (which affect only a few state variables) and the corresponding undo problems. Performance does remain a challenge in some domains though. The most important challenge, in our view, remains more effective partial undoability checking, arguably the most useful tool in practice. Towards this end, it may
be worth looking into assumption-based planning (Davis-Mendelow et al. 2013), where the planner imposes restrictions on the initial belief, identifying a solvable subset of initial states. The current incarnation pertains to conformant planning, yet an extension to contingent planning and undoability checking appears promising and feasible.

References


