Gotta Catch 'Em All! Sequence Flaws in CEGAR for Classical Planning

Martín Pozo, 1 Álvaro Torralba, 2 Carlos Linares López 1
Universidad Carlos III de Madrid, Madrid, Spain
Aalborg University, Aalborg, Denmark

Abstract
Counterexample-Guided Abstraction Refinement (CEGAR) is a prominent technique to generate Cartesian abstractions for guiding search in cost-optimal planning. The core idea is to iteratively refine the abstraction, by finding a flaw in the current optimal abstract plan. Previous works find only a single flaw, by executing the abstract plan in the concrete state space and stopping when such execution cannot be continued. We show, however, that many flaws can be identified on a single abstract plan. We introduce sequence flaws, a new definition of flaw that allows us to characterize issues in the abstract plan beyond the first one by executing the plan in a Cartesian relaxation of the problem. This greatly increases the flexibility of CEGAR regarding how to refine the abstraction. Our experiments show that across existing benchmarks a high number of sequence flaws exist in most abstract plans. We observe that the selected flaw has a high impact on the heuristic, opening research opportunities for better selection strategies.

Introduction
Abstractions are commonly employed in optimal planning to generate domain-independent admissible heuristics, as they offer great flexibility to define well-informed heuristics for the planning task at hand (Edelkamp 2001; Helmert et al. 2014; Sievers and Helmert 2021). However, such flexibility raises the question of how to efficiently compute the right abstraction. A promising method is Counterexample-Guided Abstraction Refinement (CEGAR), successfully used for Cartesian abstractions (Seipp and Helmert 2018), PDBs (Rovner, Sievers, and Helmert 2019) and domain abstractions (Kreft et al. 2023). CEGAR starts with a trivial abstraction, where all states are equivalent to each other. Then, it iteratively refines the abstraction trying to improve the heuristic value of the initial state. To do so, it searches an optimal abstract plan and executes it on the original state space. If it works, an optimal plan has been found and the task is solved. If it fails, a flaw is identified where the execution could not continue and the abstraction is refined by splitting such state, so that flaw cannot happen again (Seipp and Helmert 2013).

Recent work introduced regression flaws. Instead of executing the plan forward, the abstract plan is executed backwards from the goals. This flaw is often very different from its forward counterpart. Indeed, regression flaws greatly improve the performance of the overall procedure, resulting in more informed heuristics (Pozo, Torralba, and Linares López 2024). This shows the importance of considering new ways of computing flaws, and brings up the question whether there are other flaws that could be identified.

Indeed, CEGAR was originally introduced in the context of program verification (Ball, Podelski, and Rajamani 2001; Smaha and Hoffmann 2009; Hajdú and Micskei 2020; Löwe 2017; Albarghouthi 2015), where the notion of refinement is based on sequence interpolation, used to find flaws in several steps of the sequence. Inspired by this we consider whether the same is true in the planning setting.

In this paper, we show that multiple flaws can be identified in a single abstract plan, opening multiple alternative ways for refining the abstraction. Consider for example a problem where we need to increase two counters from 1 to 5 and an abstract plan \(\langle inc(c_1, 2, 3), inc(c_2, 4, 5), inc(c_1, 4, 5)\rangle\). Clearly, there are three separate issues with this plan: (1) \(c_1\) “jumps” from 1 to 2 before the first action; (2) \(c_1\) “jumps” from 3 to 4 before the last action; and (3) \(c_2\) “jumps” from 1 to 4 before the second action, but current methods will only find flaw (1) forward and flaw (2) backwards. Furthermore, some flaws are inherent to the abstraction and independent of the initial state and the goals. For example, if two consecutive actions \(inc(c_1, 1, 2), inc(c_1, 3, 4)\) are somewhere in the middle of the abstract plan, a flaw should be detected, regardless of the direction either before or afterwards.

We introduce sequence flaws, a new type of flaw that allows the identification of multiple issues in the same abstract plan. Our experiments show that abstract plans have many different sequence flaws that can be repaired. As a single flaw is refined, strategies to determine the flaw to select are paramount. The results support previous findings, and refining closer to the goal is usually better. But there are also promising results, and we observe that different selection strategies can sometimes lead to better heuristic functions.

Background
We consider tasks in SAS\(^+\) representation (Bäckström and Nebel 1995), where states are described in terms of a set of variables \(V\), and each \(v \in V\) has a finite domain, \(D_v\). A partial state \(p\) is a partial variable assignment over some variables \(vars(p) \subseteq V\). A (concrete) state \(s\) is a full as-
signment, i.e., $\text{vars}(s) = V$. We write $p[v]$ for the value assigned to the variable $v \in \text{vars}(p)$ in the partial state $p$.

Two partial states $p$ and $c$ are consistent if $p[v] = c[v]$ for all $v \in \text{vars}(p) \cap \text{vars}(c)$. We denote by $S(p) \subseteq S$ the set of states consistent with $p$.

A SAS* task II is a tuple $(V, O, s_0, G)$ where $s_0$ is the initial state, $G$ is a partial state that describes the goals, and $O$ is a set of operators. An operator $o \in O$ has preconditions $\text{pre}(o)$ and effects $\text{eff}(o)$, both of which are partial states, and a non-negative cost $\text{cost}(o) \in \mathbb{R}^+_0$. An operator $o$ is applicable in progression in a state $s$ if $s$ is consistent with $\text{pre}(o)$. The result of applying $o$ to $s$ is a state $s \xrightarrow{a} s'$ whenever $o$ is applicable on $s$ and $s' = s \cup \text{eff}(o)$. The post-conditions of an operator $o$ in progression are defined for $v \in \text{vars}(\text{pre}(o)) \cup \text{vars}(\text{eff}(o))$ and $\text{post}(o)[v] = \text{eff}(o)[v]$ for $v \in \text{vars}(\text{eff}(o))$ and $\text{post}(o)[v] = \text{pre}(o)[v]$ otherwise. We write $s \xrightarrow{a} s'$ as a shorthand whenever $o$ is applicable on $s$ and $s' = s \cup \text{eff}(o)$.

The state space of a task II is a transition system, $\Theta = (S, O, T, s_0, G)$, where $S$ is the set of all states, $S_G = \{s \in S \mid s$ is consistent with $G\}$ is the set of goal states, and $T = \{(s, o, s') \mid s \in S, o$ applicable in $s, s' = s \cup \text{eff}(o)\}$ is the set of transitions. A plan $\pi$ for $s$ is a sequence of operators $\langle o_1, o_2, \ldots, o_n \rangle$, s.t. the trace $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ reaches a goal state $s_n \in S_G$. The cost of $\pi$ is the sum of up cost of its operators. The goal distance from $s$ to the goal $h^*(s)$ is the minimum cost of any plan for $s$, or $\infty$ if no plan exists. A plan for $I$ is a plan for $s_0$.

A common approach to find optimal plans is to use A* search with an admissible heuristic (Dechter and Pearl 1985). A heuristic is a function $h : S \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$. The heuristic is admissible if $h(s) \leq h^*(s)$ for all $s \in S$.

Regression starts from a partial state $p$ and derives from which states we can reach some state in $S(p)$ by applying an operator (Rintanen 2008; Alcázar et al. 2013). An operator $o$ is applicable in regression in $p$ if $p$ is consistent with $\text{pre}(o) = \text{post}(o)$. The successor partial state $p'$ is defined for $\text{vars}(p) \cup \text{vars}(\text{eff}(o))$ and $\text{pre}(p') = \text{pre}(o)$ and $\text{eff}(p') = \text{eff}(o)$ for $v \in \text{vars}(\text{pre}(o))$ and $\text{eff}(p') = \text{eff}(o)$ otherwise. We use $p \xrightarrow{o} p'$ as a shorthand.

An abstraction $\alpha$ for a transition system $T = (S, O, T, s_0, G)$ is a function $\alpha : S \rightarrow S^\alpha$, where $S^\alpha$ is a finite set of abstract states. The abstract state space $\Theta^\alpha = (S^\alpha, O, T^\alpha, s_0^\alpha, G^\alpha)$ is a homomorphism of the state space, i.e., $T^\alpha = \{\alpha(s) \xrightarrow{a} \alpha(t) \mid s \xrightarrow{a} t \in T\}$, $s_0^\alpha = \alpha(s_0)$, and $G^\alpha = \{\alpha(s) \mid s \in S_G\}$. Each abstraction induces a heuristic function $h^\alpha(s)$, where $h^\alpha(s)$ is the distance from $s$ to the goal in $\Theta^\alpha$. Each abstract state $s^\alpha \in S^\alpha$ is identified with the set of states mapped to it, $S(s^\alpha) = \{s \mid s \in S, \alpha(s) = s^\alpha\}$.

Cartesian abstractions are a type of abstractions where the set of states $S(s^\alpha)$ is Cartesian $\forall s^\alpha \in S^\alpha$ (Seipp and Helmert 2018). A set of states is Cartesian if it is of the form $A_1 \times A_2 \times \cdots \times A_n$, where $A_i \subseteq D_v \forall v \in V$. Given a Cartesian set $\alpha$, we denote by $[\alpha(v)]$ the set of values that $v$ can take in $\alpha$, i.e., $[\alpha(v)] = A_i \subseteq D_v$. The intersection of two Cartesian sets is a Cartesian set, where $[\alpha'(v)] = [\alpha_1(v)] \cap [\alpha_2(v)] \forall v \in V$.

Figure 1 shows a Cartesian plan, where, for example, in $a_1$, $v_1 = \{1\}$ and $v_2 = \{2, 3\}$. Also, for any (partial) state $p$, we can build a Cartesian set $C(p)$ such that $S(C(p)) = S(p)$, by making $C(p)[v] = [C(p)[v]]$ if $v \in \text{vars}(p)$ and $C(p)[v] = D_v$ otherwise. We will use this conversion of (partial) states into Cartesian sets implicitly, so with a slight abuse of notation we define operations such as the intersection of a partial state $p$ and a Cartesian set $a$ as the Cartesian set $p \cap a := C(p) \cap a$.

The most successful technique to obtain Cartesian abstractions is CEGAR (Seipp and Helmert 2013, 2018). It starts with the trivial abstraction, which consists of a single abstract state $a$. $a[v] = D_v \forall v \in \text{vars}(v)$. Then, it is iteratively refined until reaching a termination condition or finding a concrete plan. The refinement loop finds an optimal abstract plan trace $\tau^* = a_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} a_n$, and it is executed in the concrete space, resulting in a trace $s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$. If this execution succeeds and $s_n \in S_G$, then it is an optimal plan for the task. Furthermore, we say that $\tau^*$ is mappable if each concrete state is included in the corresponding abstract state, i.e., $s_i \in S(a_i)$ for all $i \in [0, n]$.

If the abstract plan trace is not mappable, a flaw is reported and the abstraction is refined by splitting an abstract state of the plan into two, in such a way that the same flaw cannot happen again. A flaw is a tuple $(s_i, G)$ of a state $s_i \in S$ and a Cartesian set $G$. We can distinguish a different type of flaw for each reason that can cause the execution of $\tau^*$ to fail at step $i$: (1) $s_i$ is the first state in which $a_{i+1}$ is not applicable and $c$ is the set of states in $a_i$ in which $a_{i+1}$ is applicable, i.e., $c := a_i \cap \text{pre}(a_{i+1})$. (2) $s_i$ is the first state where $a_{i+1}$ is applicable but $s_i \not\in \text{pre}(a_{i+1})$ is not mapped to $a_{i+1}$. Then, $c$ is the set of states in $a_i$ from which $a_{i+1}$ is reached when applying $a_i$. (3) The sequence can be executed but $s_n$ is not a goal state. This results in the flaw $(s_n, G)$.

A flaw $(s, c)$ is repaired by splitting $\alpha(s)$ into two abstract states $d$ and $e$ with $s \subseteq d$ and $c \subseteq e$. Usually, multiple possible splits exist in different variables to fix the flaw. A split selection strategy is a criterion to choose a split among the ones that fix the flaw (Seipp and Helmert 2013, 2018). The process refines the abstraction until solving the problem either by finding an optimal plan or proving the task unsolvable (an abstract plan cannot be found). It can be stopped by some termination condition (typically a time or memory limit), resulting in an abstraction that induces a heuristic.

Recent work has introduced regression flaws, found by executing the abstract plan in regression from the goals, with better results than progression flaws (Pozo, Torralba, and Linares Lópe 2024). They are computed similarly to progression flaws, with the only difference of not requiring that the Cartesian state of the partial state is included in the abstract state but their intersection is not empty, since the former would be too restrictive. So, there are three types of flaws: (1) $p_i$ is the first partial state in regression in which $a_i$ is not backward applicable, and $c$ is the set of states in $a_i$ where $a_i$ is backward applicable. (2) $p_i$ is the first partial state in regression where $a_i$ is backward applicable but the intersection of its successor and $a_{i-1}$ is empty; then, $c$ is the set of states in $a_i$ reached when applying $a_i$ in $a_{i-1}$. (3) The sequence can be executed but $s_0 \notin \text{pre}_{p_0}$, and $c$ is the Cartesian set of $s_0$. The strength of this technique is maximizing $h$ for
Sequence Flaws

Our main contribution is the definition of sequence flaws. This allows us to find more flaws in the abstract plan.

Consider a planning task in which a worker must get a package out of a building, passing through three rooms separated by doors, all open or all closed. A button carried by the worker opens the doors of all rooms, and all doors are automatically closed when somebody leaves the building. Initially, the worker is in the first room with doors closed, and the goals are to bring the package out of the building leaving all doors open. Let $v_1$ denote the state of the doors (1 means open, 0 means closed), and $v_2$ the room where the package is located (1, 2, 3 are the rooms and 4 is the street). $o_1$ presses the button to open doors, and $o_2$, $o_3$, $o_4$ move the package, with $o_4$ moving it from (i − 1) to i. Formally,

$$V = \{v_1, v_2\} \text{ with } D_{v_1} = \{0, 1\} \text{ and } D_{v_2} = \{2, 3, 4\},$$

$$O = \{o_2, o_3, o_4\}$$

with

$$pre(o_1) = \{v_1 \rightarrow 0\}, \quad eff(o_1) = \{v_1 \rightarrow 1\},$$

$$pre(o_2) = \{v_1 \rightarrow 1, v_2 \rightarrow 1\}, \quad eff(o_2) = \{v_2 \rightarrow 2\},$$

$$pre(o_3) = \{v_1 \rightarrow 1, v_2 \rightarrow 2\}, \quad eff(o_3) = \{v_2 \rightarrow 3\},$$

$$pre(o_4) = \{v_1 \rightarrow 1, v_2 \rightarrow 3\}, \quad eff(o_4) = \{v_1 \rightarrow 0, v_2 \rightarrow 4\},$$

$$s_0 = \{v_1 \rightarrow 0, v_2 \rightarrow 1\}, \quad G = \{v_1 \rightarrow 1, v_2 \rightarrow 4\}.$$

The abstract plan shown in Figure 1 has a progression sequence flaw in $v_2$, because $o_4$ is not applicable in the state reached after applying $o_2$ (mapped to $a_1$), since the worker is not in the correct room, so that $o_4$ must be applied before $o_4$. It also has a regression sequence flaw in $v_2$ because $o_2$ is not backward applicable in the state reached after applying $o_4$ in regression (mapped to $a_1$) because the worker is not in the correct room. They are detected neither using the first progression nor regression flaw, which only detect $o_2$ is not applicable in $a_0$ because $v_1 \rightarrow 0$ (the doors are closed) and $v_1 \rightarrow 0$ instead of 1 in the goal state (doors must be opened to meet the goals). So stopping at the first flaw completely ignores the existence of problems in the other goal ($v_2 \rightarrow 4$). Sequence flaws can capture issues that are not detected until repairing all the flaws happening before them when stopping at the first flaw. In previous work, flaws are found by executing the operators in the abstract plan on the concrete (partial in regression) state space, generating a sequence $s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \ldots \xrightarrow{o_n} s_n$ ($p_0 \xrightarrow{o_n} \ldots \xrightarrow{o_0} p_0$ in regression). Instead, we consider a relaxation of such an approach, that results in a sequence of Cartesian sets.

An operator is applicable in a Cartesian set $c$ if $pre(o) \cap c \neq \emptyset$, and the result of applying $o$ to $c$ is another Cartesian set $c[\alpha]$ where $c[\alpha][v] = post(o)[v]$ if $v \in vars(post(o))$ and $c[\alpha][v] = c[v]$ otherwise. In the resulting Cartesian set, the variables of effects and preconditions of $o$ have a single value. Note that this corresponds to all states reachable by applying $o$ from any state in $c$: $S(c[\alpha]) = \{s' | s \in S(c) \land s \xrightarrow{\alpha} s'\}$. An operator $o$ is applicable in regression in $c$ if $pre(o) \cap c \neq \emptyset$, and the result of applying $o$ in regression to $c$ is another Cartesian set $regr(c, o)$ where $regr(c, o)[v] = pre(o)[v]$ for $v \in vars(pre(o))$, $\{D_0\}$ for $v \in vars(\alpha) \setminus vars(pre(o))$, and $c[v]$ otherwise. We define $c[\alpha]$ and $regr(c, o)$ as the application of an operator $o$ on a Cartesian set $c$ in progression and regression even when $o$ is not applicable.

Progression Sequence Flaws

To define progression sequence flaws, we first introduce which conditions must be fulfilled by the relaxed execution of abstract plans.

**Definition 1 (Relaxed Plan Execution).** A relaxed plan execution $r = r_0, r_1, \ldots, r_n$, for an abstract plan $\tau^a = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_2} \ldots \xrightarrow{o_n} a_n$, is a sequence of Cartesian sets $r_i$, so that:

(A) $s_0 \in S(r_0)$,

(B) if $a_{i+1}$ is applicable on $r_i$, $r_i[a_{i+1}] \cap a_{i+1} \subseteq r_{i+1}$,

(C) if $a_{i+1}$ is not applicable on $r_i$, $r_i[a_{i+1}] \setminus a_{i+1} \subseteq r_{i+1}$,

(D) $r_i \cap a_i \neq \emptyset$.

Each Cartesian set $r_i$ in a relaxed plan execution represents the states related to the corresponding abstract state $a_i$ of the abstract plan that could be reached by applying the prefix plan. The execution is relaxed, meaning that at any step in the plan, more states can be added into $r_i$ depending on the relaxation chosen. This allows, for example, the execution of a plan while ignoring some of the variables altogether to detect flaws on the remaining variables. These conditions keep the execution coherent with the application of the operators in the plan. Specifically, (A) and (B) ensure that if the abstract plan trace is executable and mappable in the concrete state space, then no flaw can be found. Condition (C) aims to keep some coherence in the execution even when an operator is not applicable. Specifically, variables not affected by the operator must keep their values, and the resulting state must satisfy the post-conditions of the operator. The intuition is that the remaining part of the execution will check if the suffix of the plan works in case it is possible to fix the prefix of the plan in such a way that the operator was applicable. If that is not the case, we consider the suffix has a flaw. Lastly, condition (D) ensures that any flaw we can find can be refitted to the corresponding abstract state $a_i$.

![Figure 1: Abstract plan with sequence flaws in $v_2$ detected neither by forward nor backward first-flaws.](image-url)
Definition 2 (Progression Sequence Flaw). Let $\tau^a = a_0 \xrightarrow{a_1} a_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} a_n$ be an abstract plan and $r = r_0, r_1, \ldots, r_n$ a relaxed plan execution for $\tau^a$. A progression sequence flaw in $\tau^a$ is a tuple $(r, i, c)$ consisting of two Cartesian sets $r_i$ and $c$ such that:

1. $a_{i+1}$ is not applicable from $r_i$, and $c$ is the set of states in $a_i$ in which $a_{i+1}$ is applicable, i.e. $c = a_i \cap pre(a_{i+1})$;
2. $a_{i+1}$ is applicable from $r_i$, but its successor does not intersect to $a_{i+1}$, i.e. $r_i \cap a_{i+1} = \emptyset$, and $c$ is the states in $a_i$ from which $a_{i+1}$ is reached by applying $a_{i+1}$;
3. $i = n$, and $r_i \cap G = \emptyset$, producing the flaw $(r_n, G)$.

Note that, to determine if there is a flaw at step $i$, only the prefix of the execution $r_0, r_1, \ldots, r_i$ is relevant, as the relaxed execution can always be continued, e.g., by setting $r_j = a_j$ for $j \in [i + 1, \ldots, n]$.

Theorem 1. Let $\tau^a = a_0 \xrightarrow{a_1} a_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} a_n$ be an abstract plan trace. Then, $\tau^a$ is mappable if $\tau^a$ has no progression sequence flaw.

Proof. On the one hand, assume $\tau^a$ has no progression sequence flaw. Then, we show by induction that the plan is mappable. For the base case, $r_0$ is the Cartesian set such that $S(r_0) = \{s_0\}$. For the inductive case, since there is no flaw of types (1) and (2), $a_0$ is applicable on $r_{i-1}$, resulting in some $r_i$ whose intersection with $a_i$ is not empty. As $S(r_i) = \{s_i\}$ then $s_i \rightarrow s_i$. Finally, as there is no flaw of type (3), $r_i \cap G \neq \emptyset$, so $s_i$ is a goal state. Hence, $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} s_n$ is a plan, where $s_i \in S(r_i)$ for $i \in [0, n-1]$ and $s_n \in S(G)$.

On the other hand, assume that $\tau^a$ is mappable. Then $s_0 \xrightarrow{a_0} \ldots \xrightarrow{a_n} s_n$ is a valid plan such that $s_i \in S(a_i)$ for all $i \in [0, n]$. Consider any arbitrary relaxed plan execution $r_0, r_1, \ldots, r_n$. By induction, $a_0$ is applicable in $r_{i-1}$ because it is applicable in $s_{i-1}$, so no flaw of type (1) exists. By condition (B) $r_i \cap [a_{i+1}] \subseteq r_{i+1}$, so $s_i \in S(r_i)$ and $r_i \cap a_i \neq \emptyset \iff \forall i \in [0, n-1]$, no flaw of type (2) exists. Finally, no flaw of type (3) exists because $s_n$ is a goal state and $s_n \in S(r_n)$.

Theorem 2. Let $\tau^a = a_0 \xrightarrow{a_1} a_1 \ldots \xrightarrow{a_n} a_n$ be an abstract plan trace. Then, $\langle o_1, \ldots, o_n \rangle$ may be a plan even if $\tau^a$ has progression sequence flaws of type (2).

Proof. The example used in (Pozo, Torralba, and Linares López 2024) applies. Consider a task with binary variables $V = \{v_1, v_2\}$, $s_0 = \{v_1 \rightarrow 0, v_2 \rightarrow 0\}$, $G = \{v_2 \rightarrow 1\}$ and operators $O = \{a_1\}$, $pre(a_1) = \{v_2 \rightarrow 0\}$ and $eff(a_1) = \{v_2 \rightarrow 1\}$, and an abstraction with states $S^a = \{a_0 = \langle \{0, 1\} \times \{0\}\}, a_1 = \{\{0\} \times \{1\}\}, a_2 = \{\{1\} \times \{1\}\}\}$. The abstract plan trace $\tau^a = a_0 \xrightarrow{a_1} a_2$ has a progression sequence flaw of type (2) in $a_1$ but $(a_1)$ is a plan, only that the final state is mapped to $a_1$ instead of $a_2$.

Even though our definition allows arbitrarily big Cartesian sets along the relaxed execution, doing so results in finding fewer flaws. In the extreme case, if all $r_i$ are completely relaxed, all operators would be applicable on $r_i$ and no flaws could be found. In Figure 1, $r_1$ could be the Cartesian set $\langle \{0, 1\} \times \{1, 2, 3, 4\}\}$. But doing this no flaw would be found in $a_4$, due to $v_2 \rightarrow 3$ is included in $r_1$, while if we make $r_1 = \langle \{1\} \times \{2\} \rangle$, we find that $a_4$ is applicable due to $v_2 \rightarrow 2$ after applying $a_2$.

Theorem 3. Let $r = r_0, r_1, \ldots, r_n$ be a relaxed execution for $\tau^a = a_0 \xrightarrow{a_1} a_1 \ldots \xrightarrow{a_n} a_n$. Let $z_i$ be a Cartesian set such that $z_i \cap a_i \neq \emptyset$, $r_{i+1} \cap a_i \in \mathbb{Z}$, and $S(z_i) \subseteq S(r_i)$. Then, there exists another relaxed execution $z = z_0, z_1, \ldots, z_{i+1}, \ldots, z_n$ such that $S(z_j) \subseteq S(r_j)$ for $j \in [i, n]$ and any progression sequence flaw of $r$ is a progression sequence flaw of z.

Proof. Any flaw found at step $i$ on $r_i$, is a flaw for $z_i$ as well:

1. If $a_{i+1}$ is not applicable on $r_i$, then there is some precondition $v_i \rightarrow x$ of $a_{i+1}$ such that $x \notin r_i[v]$. As $S(z_i) \subseteq S(r_i)$, then $x \notin z_i[v]$, and the flaw is also found in $z_i$.
2. If $S(z_{i+1}) \cap a_i \neq \emptyset$, then there exist some $v$ such that $a_{i+1}[v] \cap r_{i+1} = \emptyset$. As $S(z_i) \subseteq S(r_i)$, then $z_i[a_{i+1}[v] \cap r_{i+1}] = \emptyset$, so a flaw is found for $z$ as well.
3. If $i = n$ and $r_n \cap G = emptyset$, then $z_i \cap G = \emptyset$.

For the rest of the execution, note that applying $z_j = r_j$ for $j \in [i, n]$ results in a valid execution. However, it is worth noting that other continuations where $S(z_j) \subseteq S(r_j)$ may result in more flaws.

Progression Sequence Flaws Collection

Algorithm 1: Find Progression Sequence Flaws

```
1 flaws ← \emptyset
2 while i < n - 1 do
3    if not applicable(r, a_i) then
4        flaws ← flaws \cup \{(r, a_i \cap pre(a_i))\}
5        r ← r \cup [a_i] ; // Apply a_i even if it is not applicable
6    if r \cap a_{i+1} = \emptyset then
7        flaws ← flaws \cup \{(r, a_i \cap regr(a_{i+1}, a_i))\} // Undevalue the Cartesian set
8    for all v \in V do
9        if r[v] \cap a_{i+1}[v] = \emptyset then
10           flaws ← flaws \cup \{(r, a_i \cap regr(a_{i+1}, a_i))\}
11           i ← i + 1
12    if r \cap G = \emptyset then
13       flaws ← flaws \cup \{(r, G)\}
14 return flaws
```

Algorithm 1 shows the proposed procedure to collect all the sequence forward flaws of an abstract plan. Contrary to the standard procedure that executes the abstract plan on the concrete state space, we consider the execution over Cartesian sets. Initially, the abstraction contains a single state, and therefore the first flaw found by Algorithm 1 will be the same flaw reported by the standard procedure. However, in the next iterations Algorithm 1 continues looking for more flaws until the end of the abstract plan.
The algorithm is always invoked with the default parameters except for those special strategies introduced in the next section, so it starts at \( s_0 \). We apply operators even when they are not applicable, and when flaws of the second type occur, we “undeviate” the resulting Cartesian set. We do this by re-setting \( r_{i+1} \) to all values compliant with the abstract state \( a_{i+1} \). When a flaw with respect to \( v \) has been found, we allow \( v \) to take any value consistent with the next abstract state space. So, we might want not to report the same flaw in subsequent steps of the plan. Note that a second flaw involving the same variable \( v \) may be reported, e.g., if somewhere in the remaining abstract plan two operators are applied with contradicting preconditions over \( v \).

**Theorem 4.** All flaws returned by Algorithm 1 are progression sequence flaws.

**Proof.** Flaws are accumulated in lines 4, 7 and 13. In line 4, the flaw \( \langle r_i, a_i \cap \text{pre}(a_{i+1}) \rangle \) is added if the operator is not applicable, exactly as flaw (1) in Definition 2 does. In line 7, the flaw \( \langle r_i, a_i \cap \text{regr}(b_i, a_{i+1}) \rangle \) is added if \( r \cap b = \emptyset \), exactly as flaw (2) does. In line 13, the flaw \( \langle r_n, G \rangle \) is added if \( r_n \cap G = \emptyset \), exactly as flaw (3) does. □

Algorithm 1 does not find all forward sequence flaws. As Theorem 3 shows this would require always keeping each \( r_i \) as small as possible. Yet, there are two points in Algorithm 1 where \( r \) keeps values that could be removed in an attempt to find only flaws that are relevant. In line 5, we could replace \( r \) by \( r \cap a_{i+1} \). However, this simply insists on keeping the relaxed execution fully aligned with the abstract plan trace, which could lead to finding flaws in cases where the plan is valid through other abstract states (as Theorem 2 shows). Also, in line 9, assigning a single value from \( a_{i+1} \) instead of all of them would suffice to keep property (B). However, at that point, the algorithm has already found a flaw with respect to \( v \) so, by continuing the relaxed execution with all values in \( a_{i+1} \), we seek to only report another flaw if the execution fails from all those values.

**Regression Sequence Flaws**

A relaxed plan backward execution can be defined analogously to Definition 1 but replacing \( s_0 \) by \( G \) and the application of operators in progress by regression.

**Definition 3** (Relaxed Plan Backward Execution). A relaxed plan backward execution \( \tau = r_n, r_{n-1}, \ldots, r_0 \) for an abstract plan \( \tau^A = a_0 \xrightarrow{a_1} a_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} a_n \) is a sequence of Cartesian sets \( r_i \) so that:

(A) \( G \subseteq S(r_n) \),
(B) if \( a_i \) is regresstable on \( r_i \), \( \text{regre}(r_i, a_i) \cap a_{i-1} \subseteq r_{i-1} \),
(C) if \( a_i \) is not regresstable on \( r_i \), \( \text{regre}^L(r_i, a_i) \cap a_{i-1} \subseteq r_{i-1} \),
(D) \( r_i \cap a_i = \emptyset \).

**Definition 4** (Regression Sequence Flaw). Let \( \tau^A = a_0 \xrightarrow{a_1} a_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} a_n \) be an abstract plan and \( r = r_n, r_{n-1}, \ldots, r_0 \) a relaxed plan backward execution for \( \tau^A \). A regression sequence flaw in \( \tau^A \) is a tuple \( \langle r_i, c \rangle \) consisting of two Cartesian sets \( r_i \) and \( c \) such that:

(1) \( a_i \) is not regresstable from \( r_i \), and \( c \) is the set of states in \( a_i \) in which \( c \) is applicable, i.e. \( c = a_i \cap \text{pre}(a_i) \);
(2) \( a_i \) is regresstable from \( r_i \), but its successor does not intersect to \( a_{i-1} \), i.e. \( \text{regre}(r_i, a_i) \cap a_{i-1} = \emptyset \), and \( c \) is the states in \( a_i \) from which \( a_{i-1} \) is reached by regressing \( a_i \);
(3) \( i = 0 \), and \( s_0 \notin r_0 \), producing the flaw \( \langle r_0, s_0 \rangle \).

**Theorem 5.** Let \( \tau^A = a_0 \xrightarrow{o_1} a_1 \xrightarrow{o_2} \cdots \xrightarrow{o_n} a_n \) be an abstract plan. Then, \( \langle o_1, \ldots, o_n \rangle \) is a plan if \( \tau^A \) has no regression sequence flaw.

**Proof.** If no regression sequence flaw exists in the abstract plan, there is no flaw of type (3), so that \( s_0 \in S(r_0) \). For the inductive case, if \( s_i \in S(r_i) \) and \( r_i = \text{regr}(r_{i+1}, a_{i+1}) \), by the definition of regression there must exist \( s_{i+1} \in S(r_{i+1}) \) such that \( s_i \cap a_{i+1} = s_{i+1} \). Finally, \( s_n \in S(G) \) due to condition (A), so the sequence is a plan. □

The algorithm to collect regression flaws is like Algorithm 1 but swapping \( s_0 \) and \( G \) and using regression semantics.

Regression flaws in the state \( a_k \) are different to regression flaws in the state \( a_{k+1} \) (Pozo, Torralba, and Linares López 2024). Thus, to get all flaws, the search must be conducted in both directions.

**Flaw Selection Strategies**

Using sequence flaws allows us to identify a possibly large set of flaws for a single abstract plan. The next step is then to refine the abstraction, splitting an abstract state according to one of the flaws, so that the same abstract plan is no longer applicable in the refined abstraction. While it would be possible to refine the abstraction according to multiple flaws, it suffices to choose one of them, refining the abstraction, and finding new flaws according to the new abstract plan, as perhaps other flaws become irrelevant after the refinement. Still, with a larger set of flaws that can be chosen, repairing the right flaw at each step becomes paramount. Otherwise, finding many flaws in the abstract plan is even harmful if the chosen refinement is worse than repairing the first flaw.

We collect all flaws either in the forward (progression flaws), backward (regression flaws), or both directions (bidirectional strategies). Next, we pick one flaw according to one of the following strategies:

**Default** Choose the first flaw found along the abstract plan, the same definition of flaw used in previous work.

**Last flaw (Last)** Choose the last flaw found along the abstract plan. This is thus specially interesting for forward refinements, as flaws are found closer to the goal.

**Most refined flaw (ref)** Choose the most refined state, i.e., the one with the lowest number of values for the flawed variable respect the size of its domain. It deepens into states refined in previous steps, which results in more focused refinements. This criterion was used to choose splits among abstract plans with good results (Speck and Seipp 2022).
Highest cost operator (cost) Choose the flaw at step $i$ if $a_{i+1}$ is the operator with the highest cost among flaws. The aim is to refine at points where more cost is being spent, which could be specially relevant with cost partitioning, as the cost of many operators is low.

Causal Graph Variable Ordering (CG and CGr) Given a fixed ordering on $V$, choose the flaw related to the lowest variable in the ordering. We consider two orders based on topological order of the causal graph (Helmert 2004, 2006), which have been used before as merge strategies in merge-and-shrink heuristics (Helmert et al. 2014). CG selects first the variables with the most indirect influence over the goals, whereas the reverse ordering, CGr, attempts to select variables close to the goal first.

Iterative abstract flaws (ıt) In the forward direction, this strategy starts searching flaws at the end of the plan (Algorithm 1 parameters $r = a_n$, $i = n$) and it iteratively invokes the algorithm from the previous step of the plan until finding a flaw. Finally, if no flaw is found from the initial abstract state, then it is searched from the concrete initial state. In the backward direction, this strategy is like the first flaw but starting from the goal abstract state instead of the goal partial state, returning the first flaw from the goal partial state if no flaw is found from the goal abstract state.

Closest to goal flaw (clo) Choose the flaw closer to the goal. This is only relevant to the bidirectional case, as in the backward case this is equivalent to the default strategy, and in the forward case it is equivalent to last.

On all strategies, whenever more than one flaw could be selected, we break ties according to ref.

After selecting the flawed state $a_i$, one must decide how to split it to refine the abstraction. Typically, several possible splits exist, that divide $a_i$ into $a_i'$ and $a_i''$ according to a variable so that $r_i \cap a_i' = \emptyset$ and $c_i \cap a_i'' = \emptyset$.

The best split at each flawed state is chosen by using split selection strategies (Speck and Seipp 2022). The default strategy maximizes the amount of flaws covered, breaking ties in favor of the most refined split.

Some flaw selection strategies depend on the splits to be accurate, so the split selection strategy must use the same criterion. These strategies are ref, CG, CGr and cost. In these cases, ties are broken in favor of the most refined split. For these strategies, splits must be computed in all the flawed states before choosing one of them. Otherwise, the best split is computed only in the chosen flawed state.

To save the computations of splits, we cache the best split for each abstract state. However, the cached values are often invalidated, each time a connected abstract state is refined.

Experiments

We implemented the sequence refinement within the Scorpion planner (Seipp, Keller, and Helmert 2020). We use the Autoscale 21.11 benchmark set (Torralba, Seipp, and Sievers 2021), which contains the 42 domains of the International Planning Competitions (IPC) up to 2018 with 30 tasks each. All experiments are limited to 30 minutes and 8 GB of RAM and run in a Debian 10.2 server with an AMD EPYC 7551 CPU at 2.5 GHz.

We find optimal abstract plans using incremental search (Seipp, von Allmen, and Helmert 2020), the default and faster option in Scorpion. For single abstraction experiments we set 10 million of non-looping transitions as the termination condition because the default value of 1 million is too low for a single abstraction and for a better measurement of the performance penalty of computing sequence flaws.

In Scorpion’s implementation, goals are refined before starting the main CEGR loop as an optimization. We keep this only for forward refinements, where it improves the results, but it is disabled on all configurations using sequence flaws to compute all flaws in all steps. Another optimization, kept on all configurations, is refining all unreachable facts before goal on tasks with a single goal. They are found using the relaxed planning graph (Blum and Furst 1997). All experiments reported here use the “wanted” splitting strategy for progression flaws and the “unwanted” strategy for regression flaws, since this setting gets the best results in previous work (Pozo, Torralba, and Linares López 2024).

Code and experimental data are published in Zenodo (Pozo, Torralba, and Linares López 2024).

Single Abstraction Experiments

A comparison on the total coverage (in number of domains with more tasks being solved) of forward, backward and bidirectional strategies is shown in Table 1. $B$ solves more problems than the other strategies, and the best non-default strategy in all directions is ıt. The second-best strategies are $F_{\text{last}}$, $B_{\text{CGr}}$ and the $D_{\text{clo}}$. $F_{\text{lt}}$ solves 21 more problems.
than method $F$, it is better than $F$ in 19 domains and worse in 5 domains. Also, although the best backward strategy solves 6 fewer problems, some strategies perform better in some domains, and choosing the best strategy for each domain in any direction would solve 483 problems (32 more). So better criteria to choose flaws could solve more problems than $B$.

$GGr$ is the best strategy in many domains in all directions despite solving fewer problems in total. It favours flaws in the variables more causally related to goals, useful in blocksworld, data-network, depots, pathways, pipesworld, scannerlyze, snake and storage.

The impact of computing sequence flaws is huge, as shown in Table 2 (much larger build time). It is larger in domains with long plans like airport and agricola, since more splits must be computed in each refinement step for those. It is also larger in strategies that enlarge the abstract plan, like $ref$, and lower in strategies that do not need to compute the splits in all states: $last$ and $clo$, with the drawback of $D_{clo}$ computing and comparing flaws in both directions.

A good metric to compare heuristics is the expansions until the last $f$-layer, since this shows how good the heuristic is during the search. This is shown in Figure 2. $F_{last}$ is better than $F$ and a bit worse than $B$, since most points are above the diagonal in the first plot and below in the second one. An interesting insight is that $F_{last}$ is much better in expansions than $B_{last}$, despite solving one fewer problem. $B_{cost}$, the second-best backward sequence strategy, is very similar in expansions to $B$, so the difference in coverage is mostly caused by the larger time required to build abstractions.

Table 2 shows statistics to analyze the behaviour of each strategy. Per-domain details are omitted for space reasons.

One interesting observation is how many times the cost of the abstract plan increased, since this describes the preferences of each strategy for refining states: strategies that increase the cost of the abstract plan many times are focused on getting the actual plan, while strategies with few increments are focused in increasing the $h$ value of other states. $ref$ improves the cost more often in almost all domains, though it gets a lower total improvement due to parexprinter. Results vary on the domain, but $clo$ has the largest number of increments for progression flaws, and $G$ and $cost$ get more improvements than $last$, $clo$ and $GGr$. $ref$ tries to increase the plan length while $last$, $clo$ and $GGr$ perform width-like refinements. $it$ refines close to the goal but enlarging the plan, so its refinements are very useful. A similar behaviour is observed in the tasks solved during the loop, where $F$, $F_{ref}$, $B_{last}$ and $D_{cost}$ are the best strategies, but it is not equivalent because refining closer to $s_0$ is more relevant for this than increasing the cost.

Another interesting point is the total number of refinements, an indicator of the number of states of the abstraction and a proxy for the density of transitions because the loop
ends when it has 10M non-looping transitions. The results vary on the domain, but cost, ref and CG are the strategies with the highest number of refinements, while CG, last and clo are the strategies with fewest refinements.

Two related features are the total number of flaws and the percentage of states with a flaw in an abstract plan. The first one is correlated with the length of the plan, as the more states in the abstract plan, the more flawed states can exist. The second feature is inversely correlated to the former, as the shorter the plan, the more likely it is to have flaws in a larger percentage of the states. ref has the largest number of flawed states and the lowest percentage of flawed states because its plans are the longest ones. But the behaviour per domain varies, and CG has a low number of flawed states overall but many more flaws than the rest in elevators, hiking, micomic and rovers. CG, last and clo have the largest percentage of flawed states due to their shorter plans.

The last analysis is the relative position of the refined state with respect to the plan length. That is, 0% means the initial state and 100% means the goal state, while in a plan of 3 states the state of the middle would be the 50%, and so on. The average in the total of domains is dominated by $F_{1s}$, though $F_{last}$ and $D_{clo}$ are close and they win in some domains. It may seem strange that $D_{clo}$ refines states less close to the goal, but the abstract plans found in each iteration depend on previous refinements, so refining a state closest to the goal can lead to not finding flaws as close in the next steps. The values seem low but, as their plans are short, not finding flaws in the last states greatly decreases the average.

Additive Abstraction Experiments

Table 3 shows in how many domains each strategy performs better and the coverage for additive abstractions via saturated cost partitioning (Seipp, Keller, and Helmert 2020). The best strategy is $D_{ref}$, which solves 4 more problems than $B^{add}$ and 14 more problems than $F^{add}$. $D_{clo}$ solves only one problem less, and it is better than other strategies in more domains. $B^{add}$ solves the same number of problems, and it is better than $B^{add}$ in 2 domains and worse in only one. $F^{add}$ is the best forward strategy, solving 12 more problems than $F^{add}$ and even 2 more problems than $B^{add}$.

507 tasks can be solved by choosing the best forward strategy in each domain, despite being worse separately.

Sequence flaws are more useful in additive abstractions than in a single abstraction, as regression flaws turned out to be not so good in partitioned problems.

No solution is found in the loop because the problem is partitioned, but all other statistics can be compared. The time to build abstractions is very low because tasks and transition limits are smaller. Overall, all features are lower, despite getting better heuristics. The behaviour among strategies is like for a single abstraction with small differences.

Related Work

Model Checking is an area which aims to automatically verify the correctness of programs. In symbolic model checking, the transition relation of a system is represented with a first-order logic formula. A program is incorrect if the error location is reachable (McMillan 2005; Vizel and Grumberg 2009). CEGAR is one of the most successful techniques in Model Checking, and it was the inspiration for the use of CEGAR in Classical Planning (Hajdu and Micskei 2020; Löwe 2017; Albarghouthi 2015; Seipp and Helmert 2013).

Our work is inspired by sequence interpolation approaches in the context of model checking (McMillan 2005; Albarghouthi 2015). Craig interpolants are a well-known technique to prove the unreachability of a formula in symbolic model checking. Given a pair of formulas $(A, B)$, such that $A \land B$ is inconsistent, an interpolant for $(A, B)$ is a formula $A$ such that $A$ implies $A$, $A \land B$ is unsatisfiable, and $A$ refers to the common symbols of $A$ and $B$ (McMillan 2005).

This idea can be generalized to sequences of formulas. Given a sequence of formulas $\Gamma = A_1, \ldots, A_n$, $\hat{A}$, $\hat{B}$ is an interpolant for $\Gamma$ when (i) $A_0 = \top$ and $A_n = \bot$ and (ii) for all $1 \leq i \leq n$, $A_{i-1} \land A_i$ implies $A_i$ and (iii) for all $1 \leq i < n$, $A_i \in (\mathcal{L}(A_1, \ldots, A_i) \cap \mathcal{L}(A_{i+1}, \ldots, A_n))$.

Note the similarity of sequence interpolants and sequence flaws: $r_i$ Cartesian sets are similar to $A_i$, and sequence flaws would be the equivalent to interpolants. But significant differences exist, as we execute the abstract plan in a relaxed way so our flaws are incomparable to such interpolants.

In the context of planning, some works explore how to conduct CEGAR in different ways (Seipp and Helmert 2018; Eifler and Fickert 2018), or how to combine multiple abstractions via cost partitioning (Seipp 2017). The closest work is the refinement strategies by Speck and Seipp 2022, which also identifies multiple flaws, but focusing on finding flaws from multiple abstract plans instead of only one.

Another topic in planning with similarities to our work is Partial-Order Causal-Link planning (Penberthy and Weld 1992; Younes and Simmons 2003; Bercher 2021), which refines partial plans by detecting flaws in them. In this case, flaws can be an open precondition not protected by a causal link or an operator that can delete a precondition before it is needed. Open preconditions are fixed adding a causal link and threats are fixed moving the operator before the variable is produced or after the operator that needs it. Multiple flaws exist at each step of the refinement loop and the flaw selection is critical to reduce the steps required to get a correct plan, but the flaws and the refinements are different.

Conclusions

CEGAR is a method to iteratively refine abstractions by identifying flaws in optimal abstract plans. But previous work find a single flaw per plan, the first one along its execution. Our main contribution is a new type of flaw that allows searching flaws after the first one. This enables identifying flaws that could not be found otherwise, and it opens research opportunities for new refinement strategies.

We have experimentally shown that different selection flaw strategies result in very different behaviour, and that each strategy is better than the others in some domains. We have also shown that iterative strategies and strategies based on the most refined state can get better heuristics than regression flaws, especially in additive abstractions, opening research opportunities for smarter flaw selection strategies.
References


