A Verified Compositional Algorithm for AI Planning

Mohammad Abdulaziz
Technical University of Munich, Germany
mohammad.abdulaziz@in.tum.de

Charles Gretton
Australian National University, Australia
charles.gretton@anu.edu.au

Michael Norrish
Data61, CSIRO, Australia
michael.norrish@data61.csiro.au

Abstract

We report on our HOL4 verification of an AI planning algorithm. The algorithm is compositional in the following sense: a planning problem is divided into multiple smaller abstractions, then each of the abstractions is solved, and finally the abstractions' solutions are composed into a solution for the given problem. Formalising the algorithm, which was already quite well understood, revealed nuances in its operation which could lead to computing buggy plans. The formalisation also revealed that the algorithm can be presented more generally, and can be applied to systems with infinite states and actions, instead of only finite ones.

Our formalisation extends an earlier model for slightly simpler transition systems, and demonstrates another step towards formal treatments of more and more of the algorithms and reasoning used in AI planning, as well as model checking.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Computing methodologies → Planning for deterministic actions; Computing methodologies → Planning with abstraction and generalization; Software and its engineering → Software verification

Keywords and phrases AI Planning, Compositional Algorithms, Algorithm Verification, Transition Systems

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.18

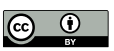
Supplement Material All of our HOL4 scripts are available online at `10.5281/zenodo.3298914`.

Funding *Mohammad Abdulaziz*: This author is supported by the DFG Koselleck Grant NI 491/16-1.

1 Introduction

State spaces of problems in fields such as artificial intelligence (AI) planning and model checking can be modelled as digraphs, where vertices and edges represent states and transitions, respectively. Explicitly representing such state spaces is infeasible in realistic systems. Instead, the digraph modelling the state space is described with a *propositionally factored* representation, using languages such as STRIPS by Fikes [17] or SMV by McMillan *et al.* [27]. We work in the space of tools and algorithms for solving problems represented in this way.

When working with such factored representations, controlling the *state space explosion* is critically important. A powerful, general approach to this problem is the *compositional* approach. Here, a solution to a problem instance is found, or approximated, by composing solutions to one or many (possibly exponentially) smaller derived sub-problems, or "abstractions". Practically, this approach is one of the few known feasible approaches to solve problems concerning the state space of the given factored system. This is because it avoids



constructing and performing computations on the large digraph modelling the state space, and only constructs and processes abstracted state spaces.

A planning problem is a *reachability* problem in a digraph representing the state space: given an initial state, is it possible to construct a sequence of actions that reaches a goal state? AI planning has many applications, including safety-critical ones, such as aerospace applications [34, 35]. Thus, it would be of great utility to use formal methods to increase the reliability of AI planning software, techniques, and frameworks. Indeed, this was realised by many early authors who used formal methods in AI planning applications [9]. However, all prior work was limited to using model checking techniques to formally verify planning domain model properties and plan properties, and none of the previous authors embarked on verifying a planning algorithm. In this paper we present the first formal verification of a planning algorithm. We use HOL4 [33] to formally verify the correctness of a compositional planning algorithm, which we published earlier [5], showing that the algorithm is indeed correct. One might wonder: why use a theorem prover to verify the algorithm, instead of a model checker like earlier applications of formal methods to planning? This is due to (i) the complexity of verifying a planning algorithm compared to verifying properties of planning models and plans as in earlier work, and (ii) the limitations of model checking formalisms, which are inadequate for representing the algorithm, let alone verifying it. Also, HOL4 has a transition systems theory library suitable to reasoning about planning algorithms [6].

The algorithm we verify works by dividing a planning problem into multiple isomorphic abstractions, solving each of those abstractions separately, and finally composing those solutions in a solution to the concrete problem. Each abstraction is an under-approximation of the problem that is isomorphic to a *descriptive quotient* (hereafter, quotient) of the problem. In our earlier work, this quotient was computed based on symmetries in the planning problem. This earlier work empirically established that this algorithm performs extremely well on benchmark planning problems which have symmetries.

As experienced practitioners might expect, formalisation in a theorem prover yields concrete benefits. In our case, we (i) gain a precise (and hitherto unappreciated) characterisation of what we required of the planning algorithm that solves the generated sub-problems; (ii) fix our algorithm to remove our dependency on that assumption; (iii) extend the algorithm's applicability to problems whose state variables are of arbitrary types, and not necessarily Boolean, thus showing its applicability to numerical and hybrid planning; and (iv) we prove its validity for a more general class of quotients, quotients which are not necessarily computed using problem symmetries.

Finally, we note that elements of our formalisation can be easily modified to accommodate the compositional model-checking algorithm by Ip and Dill [22, 23], which is used to perform model checking on systems with multiple isomorphic components in the Murphi verification system.

Contributions:

- We provide formal definitions of the notion of planning problems and develop a theory library concerning them (Section 2.2). This is a substantial extension of an existing HOL4 library on factored transition systems which was developed to verify algorithms to compute upper bounds on transition system state space diameters [1, 2, 3, 6].
- We formally define a state-of-the-art planning algorithm for planning, namely planning via descriptive quotients, that is used for efficiently solving planning problems with symmetries.

- We develop a significant theory to establish the correctness of the connection between the abstracted sub-problems and the original. In particular, we must answer two questions:
  - when and how can sub-plans that solve (abstracted) sub-problems be concatenated to solve the original concrete problem's goal?
  - how should a descriptive quotient solution be instantiated—i.e.,"lifted back" to the level of the original concrete problem—so as to create multiple plans for solving the symmetric sub-problems of the original?
- We believe our work is the first verification of a symmetry-breaking technique or a quotient-based technique for problems on transition systems. Our verification forced us to identify an important assumption about the behaviour of the planner used to solve the abstracted sub-problems.

2 Preliminaries

2.1 Standard HOL4 Types and Operations

HOL4 provides a rich library of operations over standard types such as lists and sets, giving a powerful combination of facilities from mathematics and functional programming. Here, we briefly describe those that we use below.

In the theory of lists: lists are either empty ("nil") written $[\,]$, or a head element h followed by the rest of the list t, written $h :: t$. We write $l_1 \mathbin{+\!\!+} l_2$ to represent the concatenation of the lists l_1 and l_2. Lifting this to lists of lists, we write $\mathsf{FLAT}\ ll$ to mean the concatenation of all the lists contained within ll. We write $\mathsf{MEM}\ e\ l$ to mean that e is an element of list l. More generally, we can denote the set of all the elements contained in a list by writing $\mathsf{set}\ l$. Finally, we can write $\mathsf{MAP}\ f\ l$ to represent the pointwise application of function f to all elements of the list l, returning a list of equal length, but with elements possibly of a different type.

Most of the set notation we use should be familiar. Apart from set comprehensions and standard operators such as union and intersections, we also write $f(\!|x|\!)$ to mean the image of set x under function f, and f^{-1} for f's inverse (taking care to only use this when f is a bijection on the relevant sets).

We make extensive use of the HOL4 theory of finite maps, which are functions whose domains are finite. The domain of a map f is written $\mathcal{D}(f)$. Applying a map f to a domain element d is written $f\ `\ d$. We write $f \subseteq g$ to mean that map f is a submap of g – i.e., f and g agree on all elements in $\mathcal{D}(f)$. Finally, we can combine two maps, writing $f \uplus g$. If the maps f and g have overlapping domains, the result takes elements in the overlap to f's values (the union "biases left").

Below, all statements appearing with a turnstile ($\vdash$) are HOL4 theorems, automatically pretty-printed to LaTeX, and using this notation.

2.2 Factored Transition Systems in HOL4

We now review basic concepts about propositionally factored representations of transition systems and how they are formalised in HOL4. The distinctive feature of these represenations is that sets of edges are compactly described in terms of "actions". This representation is equivalent to representations commonly used in the AI planning and model checking communities (e.g. STRIPS [17] and SMV [27, 13]).

▷ Definition 1 (States and Actions). A state, x, is a finite map from variables to values, i.e. a finite set of mappings $v \mapsto b$, where v is a variable and b is a value. An action is a

pair of finite maps, (p, e), where p represents the *preconditions* and e represents the *effects*. The domain of an action is the union of the domains of its preconditions and effects, i.e. $\mathcal{D}(\pi) \equiv \mathcal{D}(p) \cup \mathcal{D}(e)$, for $\pi = (p, e)$. (Note how we are overloading/extending the syntax for the domain of a finite map ($\mathcal{D}(fm)$) to also mean the domain of an action ($\mathcal{D}(\pi)$), and (below) the domain of a system.)

▷ Definition 2 (Factored System). A propositionally factored system, δ, is a set of actions. We write $\mathcal{D}(\delta)$ for the domain of δ, which is the union of the domains of all the actions in δ.

To make the types explicit, a propositionally factored system in HOL4 has states as finite maps $\alpha \mapsto \beta$ (polymorphic in both domain (α) and codomain (β)).[1] An action is then a pair of such states $(\alpha \mapsto \beta) \times (\alpha \mapsto \beta)$, and a factored transition system δ is a set of such actions.

The valid states of a system δ, written $\mathbb{U}(\delta)$, are those that have the same domain as the system:

$$\mathbb{U}(\delta) \stackrel{\text{def}}{=} \{ x \mid \mathcal{D}(x) = \mathcal{D}(\delta) \}$$

The valid plans of a system (δ^*) are those composed of actions drawn from δ:

$$\delta^* \stackrel{\text{def}}{=} \{ \vec{\pi} \mid \mathsf{set}\ \vec{\pi} \subseteq \delta \}$$

▷ Definition 3 (Execution). When an action (p, e), denoted by π, is executed at state x, it produces a successor state $\mathsf{ex}(x, \pi)$, formally defined as $\mathsf{ex}(x, \pi) = \mathsf{if}\ p \subseteq x\ \mathsf{then}\ e \uplus x\ \mathsf{else}\ x$. We lift ex to lists of actions $\vec{\pi}$ as the second argument. So $\mathsf{ex}(x, \vec{\pi})$ denotes the state resulting from successively applying each action from $\vec{\pi}$ in turn, starting at x, which corresponds to a path in the state space. In HOL4 action execution and action sequence execution are defined as follows:

$$\mathsf{state\text{-}succ}\ x\ (p,e) \stackrel{\text{def}}{=} \mathbf{if}\ p \sqsubseteq x\ \mathbf{then}\ e \uplus x\ \mathbf{else}\ x$$

$$\mathsf{ex}(x,\pi::\vec{\pi}) \stackrel{\text{def}}{=} \mathsf{ex}(\mathsf{state\text{-}succ}\ x\ \pi,\vec{\pi})$$
$$\mathsf{ex}(x,[\,]) \stackrel{\text{def}}{=} x$$

The result of executing an action (p,e) on a state x depends on whether the preconditions of the action are satisfied by the state or not, which is modelled by the $p \sqsubseteq x$ relation. If the state satisfies the preconditions, then the state resulting from the execution is the same as the original state, but amended by the effects of the executed action. Otherwise, the result of the execution does not affect a change to the state. The finite map union operation, $e \uplus x$, models amending the state by the action effects e.

Our formal definition of action execution follows that from our earlier paper [5]. Having a total execution function (as above) is somewhat unusual for classical deterministic planning. The choice is more typical in robotics, and in settings where automated planning is undertaken under uncertainty. For example, the *de facto* standard in robotic planning is to plan in a partially observable Markov decision process [20, 10], in which the robot cannot generally know for sure if an action will have an effect or not. However, as machine learning becomes increasingly pervasive, both in the task of learning system models [8, 31], and in the task of computing plans [38], we can expect it to become increasingly common place for planned actions in classical deterministic settings to have no effect, since learning agents tradeoff model accuracy for model complexity. In addition to the totality of our definition of execution,

[1] To model STRIPS or SMV transition systems, β would be instantiated with *bool*.

we note that our definition is more general than other formalisms as it allows an action to execute in a state when the action is defined using symbols that are not part of that state. These properties of our definition made our proofs smoother, and helped us derive more general theorems.

A sanity check of our execution semantics is the following theorem, which states that the result of executing a valid action sequence on a valid state is also a valid state.

$$\vdash \vec{\pi} \in \delta^* \wedge x \in \mathbb{U}(\delta) \Rightarrow \mathsf{ex}(x, \vec{\pi}) \in \mathbb{U}(\delta)$$

When the codomain of a state is Boolean, we give examples of states and actions using sets of literals. For example, $\{v_1, \overline{v_2}\}$ is a state where state variables v_1 is (maps to) true, and v_2 is false and its domain is $\{v_1, v_2\}$. $(\{v_1, \overline{v_2}\}, \{v_3\})$ is an action that if executed in a state where v_1 and $\overline{v_2}$ hold, it sets v_3 to true. $\mathcal{D}((\{v_1, \overline{v_2}\}, \{v_3\})) = \{v_1, v_2, v_3\}$.

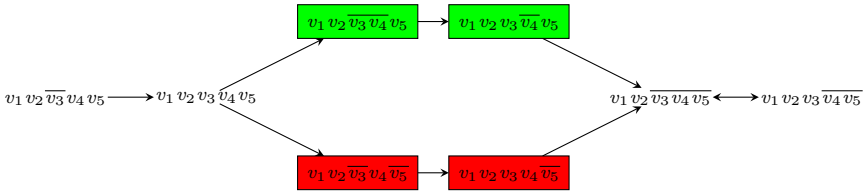


Figure 1 The largest connected component of the state space of the problem from Example 2. It shows the presence of symmetries between different states.

▷ Example 1. An example factored system δ is $\{\pi_1, \pi_2, \pi_3\}$, where the actions π_1, π_2 and π_3 are defined as $(\emptyset, \{v_3\})$, $(\{v_1, v_3\}, \{\overline{v_3}, \overline{v_4}\})$, and $(\{v_2, v_3\}, \{\overline{v_3}, \overline{v_5}\})$, respectively. The largest connected component of its state space is shown in Figure 1.

Note that, unlike in our original algorithm [5], the codomain of states is not restricted to be *bool* since a lot of the theory we develop here applies to factored systems regardless of the codomain of the state. Indeed, because we do not restrict the codomains to *bool* we are able to prove that the algorithm verified here can be used for planning problems with infinite states.

▷ Definition 4 (Planning Problem). A planning problem Π is a 3-tuple $\langle I, \delta, G \rangle$, with I the initial state of the problem, G a partial state representing a set of goal states, and δ a set of actions. We define the domain of the problem, $\mathcal{D}(\Pi)$, to be domain of its actions, $\mathcal{D}(\delta)$. The set of valid states, written $\mathbb{U}(\Pi)$, with respect to a planning problem Π, corresponds to the set $\mathbb{U}(\delta)$. In HOL4, we formalise this as a record type:

```
(α, β) planningProblem = <|
    I : α ↦ β;
    δ : (α ↦ β) × (α ↦ β) → bool;
    G : α ↦ β
|>
```

Problem Π is *valid* if the initial state is a valid state and the goal describes an assignment constraint on a subset of the problem's domain. In HOL4:

$$\textsf{valid-prob}\ \Pi \stackrel{\text{def}}{=} \Pi.I \in \mathbb{U}(\Pi.\delta) \wedge \mathcal{D}(\Pi.G) \subseteq \mathcal{D}(\Pi)$$

Henceforth, we will work only with valid problems. We refer to the initial state, actions or goal of problem Π as $\Pi.I$, $\Pi.\delta$ or $\Pi.G$ respectively. We may also omit the Π if it is clear from the context, e.g. I for $\Pi.I$ and δ_i for $\Pi_i.\delta$.

Finally, an action sequence $\overrightarrow{\pi}$ is a plan/solution for a planning problem Π iff that sequence is valid, and if all goal assignments are present in the state reached by executing that action sequence from the initial state:

$$\Pi \text{ solved-by } \overrightarrow{\pi} \stackrel{\text{def}}{=} \overrightarrow{\pi} \in \Pi.\delta^* \wedge \Pi.G \sqsubseteq \mathsf{ex}(\Pi.I, \overrightarrow{\pi})$$

▷ Example 2. An example planning problem is Π_1 with $\Pi_1.I \equiv \{v_1, v_2, v_3, v_4, v_5\}$, $\Pi_1.G \equiv \{\overline{v_4}, \overline{v_5}\}$, and actions $\Pi_1.\delta$ assigned to be δ from Example 1. The state space of that problem is that of the factored system δ, which represents its actions. A solution to that problem is the action sequence $[\pi_1; \pi_2; \pi_1; \pi_3; \pi_1]$.

2.3 Motivating Planning via Descriptive Quotient

Better scalability is the core motivation for planning using a descriptive quotient. The algorithm treats the situation where a concrete problem can be decomposed into a set of isomorphic sub-problems. We need only find a solution for one sub-problem, and then it is a simple matter of instantiating that solution for each problem in the series to arrive at a solution for the concrete problem. These ideas can be made clear if we consider the GRIPPER problem, which happens to be a benchmark problem of the *International Planning Competition* [26]. A robot with left and right grippers must move a set of N indistinguishable packages from a common *source* location to a common *destination*. The left and right grippers are symmetric, because if we changed their names, by interchanging the terms "left" and "right" in the problem description, we are left with an identical problem. Packages are also interchangeable, and symmetric in this sense. The descriptive quotient here describes the problem of moving one package with one gripper to the destination, and then returning the robot to the source location with its gripper unencumbered. A plan for the quotient represents a solution for a part of the gripper problem, for one package. If we instantiate that quotient plan to move each package, and concatenate the instantiated plans, we arrive at a plan for the concrete problem. Some first package is moved to the goal, then a second, a third, and so on until all packages are in their goal location, and the problem is solved.

When in use, and compared to other planning algorithms, the algorithm we study here comes with some overhead. Specifically, it has four inputs, and only the first of which is common to all planning algorithms. These are: (i) a planning problem, (ii) an under-approximation of that problem, also known as the descriptive quotient, (iii) a plan for the descriptive quotient and (iv) a set of instantiations of the descriptive quotient. The last three objects are peculiar to the algorithm we investigate and are computed from the first input in a preprocessing step, based on symmetries in the given planning problem [5]. After this preprocessing step, a plan is calculated for the descriptive quotient problem which is usually much smaller than the concrete problem at hand. Then, the quotient's solution is instantiated to solve sub-problems of the given problem. Lastly, those sub-problem solutions are concatenated to form a solution for the entire problem.

The primary strength of planning via descriptive quotient is that the state space of a quotient is small relative to that of the concrete problem. This algorithm is thus relatively efficient at planning compared to an algorithm that searches for a plan in the state space of the concrete problem. However, is it effective compared to other methods that exploit symmetries for planning? The state-of-the-art method to break symmetries for planning is *orbit search* [30]. That method exploits the fact that a symmetry between state variables

induces a symmetry between states. Orbit search exploits that during the search for a plan since one only needs to visit one state out of every set of symmetric states. However, pruning the state space that way still gives rise to a search space that could be exponentially larger than the descriptive quotient's state space. For instance, the descriptive quotient of a GRIPPER problem with 20 packages is solved by breadth-first search expanding only 6 states [5]. On the other hand, a state-of-the-art system implementing orbit search reports expanding $60K$ states solving the same GRIPPER problem [30]. This difference is highlighted in the next example.

▷ Example 3. In the problem Π_1 from our example earlier, state variables v_4 and v_5 are symmetric, i.e. Π_1 would stay the same if we permute them. Also v_1 and v_2 are symmetric. This variable symmetry induces symmetries between states as shown in Figure 1, where the two green states are symmetric with the red ones, i.e. permuting them does not change the state space. Ideally, the orbit search method would construct a state space where symmetric states are contracted as the one shown in Figure 3, which is clearly smaller in size than the original state space in Figure 1.[2]

On the other hand, following our previously published algorithm, a descriptive quotient, Π'_1, of the problem Π_1 is computed by replacing every variable in Π with a symbol, where symmetric variables are replaced with the same symbol. Thus, Π'_1 has initial state, actions and goals that are $\{p_1, p_2, p_3\}$, $\{(\{p_1, p_2\}, \{\overline{p_2}, \overline{p_3}\}), (\emptyset, \{p_2\})\}$, and $\{\overline{p_3}\}$, respectively. The largest component of the state space of Π'_1 is shown in Figure 2. It is clearly smaller than the original state space shown in Figure 1, as well as the state space constructed by orbit search shown in Figure 3.

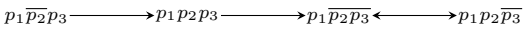


Figure 2 The largest connected component in the state space of the descriptive quotient of the problem in Example 2.

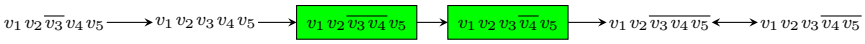


Figure 3 The state space which the orbit search algorithm could construct and in which it would search for a solution to the problem from Example 2. In the case that there were multiple symmetric states in the original problem, here only one *canonical* state from that set appears.

3 Sub-Plan Concatenation

The algorithm we describe and verify synthesises a concrete plan by concatenating a series of sub-plans. Each sub-plan solves one sub-problem of the concrete problem at hand. The first step of formally develop these ideas is describing sufficient conditions which enable one to synthesise a concrete plan according to a concatenation operation.

[2] For a comprehensive description of orbit search planning consult Pochter et al. [30].

3.1 Needed Assignments

To concatenate plans safely, the algorithm needs to constrain states encountered between the execution of two concatenated plans to be compatible with the resources that might be used by a plan for the second problem. Compatibility is guaranteed if the intermediate state is consistent with the *needed assignments* of the second sub-problem. To understand this concept, suppose we have a plan $\vec{\pi}$ for a planning problem Π. What can we change in $\Pi.I$ and still guarantee that $\vec{\pi}$ solves the amended problem? Needed assignments are those assignments in $\Pi.I$ which cannot be changed.

▷ Definition 5 (Needed Assignments). Needed assignments, $\mathcal{N}(\Pi)$, are assignments in the preconditions of actions and goal conditions that also occur in I, i.e., $\mathcal{N}(\Pi) = (\mathsf{pre}(\delta) \cap I) \cup (G \cap I)$, where $\mathsf{pre}(\delta) \equiv \bigcup\{p \mid (p,e) \in \delta\}$. Formally, we begin by characterising a planning problem's needed variables, those state variables that shall be the subject of needed assignments:

$$
\begin{array}{l}
\mathcal{D}(\mathcal{N}(\Pi)) \stackrel{\text{def}}{=} \\
\quad \{\, v \mid \\
\qquad v \in \mathcal{D}(\Pi.I) \wedge \\
\qquad ((\exists p\ e.\ (p,e) \in \Pi.\delta \wedge v \in \mathcal{D}(p) \wedge p \mathbin{`} v = \Pi.I \mathbin{`} v) \vee \\
\qquad\quad v \in \mathcal{D}(\Pi.G) \wedge \Pi.I \mathbin{`} v = \Pi.G \mathbin{`} v)\}
\end{array}
$$

Then, the set of needed assignments associated with a problem Π is:

$$\mathcal{N}(\Pi) \stackrel{\text{def}}{=} \Pi.I\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi))}$$

where $x\!\downarrow_{vs}$ denotes the state x restricted/projected to assignments to variables vs.

▷ Example 4. For Π_1 from our earlier example, we have that $\mathcal{N}(\Pi_1) = \{v_3, v_1, v_2\}$.

We are then able to prove the following sanity-check theorem:

▷ Proposition 1. For problem Π, a plan $\vec{\pi}$ will work from *any* state x that provides the needed assignments of that problem, even if x disagrees with the initial state of the problem on the assignments to some other—i.e. *not-needed*—state variables.

$$
\begin{array}{l}
\vdash \mathsf{valid\text{-}prob}\ \Pi \wedge \mathcal{N}(\Pi) \sqsubseteq x \wedge \mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), \vec{\pi}) \Rightarrow \\
\qquad \Pi\ \mathsf{solved\text{-}by}\ \vec{\pi} \Rightarrow \Pi.G \sqsubseteq \mathsf{ex}(x, \vec{\pi})
\end{array}
$$

The assumption $\mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), \vec{\pi})$ says that if $\vec{\pi}$ is executed from $\mathcal{N}(\Pi)$, the preconditions of all actions in $\vec{\pi}$ shall be satisfied.

3.2 Concatenating Two Plans

Suppose we have a plan for each of two given problems. We now establish a core condition that, if satisfied, allows us to concatenate those plans to obtain an execution that satisfies the goal conditions of both problems. It may be that some state variables are common to both problems. We shall then require that for some total ordering of the problems, the preceding problem goal includes the needed assignments of the succeeding problem. Formally, we have the *preceding problem* relation:

▷ Definition 6 (Preceding Problems).

$$\Pi_1 \lhd \Pi_2 \stackrel{\text{def}}{=} \Pi_1.G\!\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))} = \mathcal{N}(\Pi_2)\!\downarrow_{\mathcal{D}(\Pi_1)} \wedge \Pi_1.G\!\downarrow_{\mathcal{D}(\Pi_2)} = \Pi_2.G\!\downarrow_{\mathcal{D}(\Pi_1)}$$

In words, (i) The needed assignments of Π_2 which a plan for Π_1 could possibly affect occur in G_1, and (ii) G_2 contains all the assignments in G_1 which a plan for Π_2 could affect.

▷ Example 5. Consider a problem Π_2 s.t. $\Pi_2.I \equiv \{\overline{v_4}, \overline{v_5}, v_6\}$, $\Pi_2.\delta \equiv \{(\{\overline{v_5}\}, \{v_4, \overline{v_6}\}), (\{\overline{v_4}\}, \{v_5, \overline{v_6}\})\}$ and $\Pi_2.G \equiv \{\overline{v_4}, \overline{v_5}, \overline{v_6}\}$, respectively. $\mathcal{N}(\Pi_2) = G_1 = \{\overline{v_4}, \overline{v_5}\}$. Since $G_1{\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))}} = G_1$, $(I_2{\downarrow_{\mathcal{D}(\Pi_1)}}){\downarrow_{\mathcal{D}(\mathcal{N}(\Pi_2))}} = G_1$, $G_1{\downarrow_{\mathcal{D}(\Pi_2)}} = G_1$ and $G_2{\downarrow_{\mathcal{D}(\Pi_1)}} = G_1$, we have $\Pi_1 \lhd \Pi_2$.

Our precedence relation guarantees the following two properties.

▷ Proposition 2. If a planning problem Π_1 precedes another planning problem Π_2, i.e. $\Pi_1 \lhd \Pi_2$, then a plan for Π_1 always preserves the needed assignments of Π_2.

$$\vdash \Pi_1 \lhd \Pi_2 \wedge \Pi_1.G \sqsubseteq \mathsf{ex}(x, \vec{\pi}) \wedge \vec{\pi} \in \Pi_1.\delta^* \wedge \mathcal{N}(\Pi_2) \sqsubseteq x \wedge \textsf{valid-prob}\ \Pi_1 \Rightarrow \mathcal{N}(\Pi_2) \sqsubseteq \mathsf{ex}(x, \vec{\pi})$$

▷ Proposition 3. If $\Pi_1 \lhd \Pi_2$, then the a plan for Π_2 does not invalidate a goal of Π_1.

$$\vdash \Pi_1 \lhd \Pi_2 \wedge \Pi_2.G \sqsubseteq \mathsf{ex}(x, \vec{\pi}) \wedge \vec{\pi} \in \Pi_2.\delta^* \wedge \textsf{valid-prob}\ \Pi_2 \wedge \Pi_1.G \sqsubseteq x \Rightarrow \Pi_1.G \sqsubseteq \mathsf{ex}(x, \vec{\pi})$$

3.3 Concatenating Many Plans

The above analysis can be leveraged now to understand the situation where we have plans for many problems, and where a concatenation of those plans achieves and maintains goal conditions for all problems. We shall suppose that the set of problems are totally ordered according to our precedence relation.

▷ Lemma 1. Consider a sequence $\Pi_1 \ldots \Pi_N$ satisfying $\Pi_j \lhd \Pi_k$ for all $1 \leq j < k \leq N$, and a state x that satisfies the initial state of every problem Π_i, for $1 \leq i \leq N$. For $1 \leq i \leq N$ let $\vec{\pi}_i$ be a plan for Π_i for which $\textsf{sat-pre}(\mathcal{N}(\Pi_i), \vec{\pi}_i)$ holds. Then, not only is each $\vec{\pi}_i$ a plan for Π_i, but executing the entire concatenation $\vec{\pi}_1 +\!\!+ \vec{\pi}_2 +\!\!+ \ldots \vec{\pi}_N$ from x *also* satisfies the goals of each Π_i.

$$\begin{array}{l}
\vdash \lhd_l\ \Pi_l \Rightarrow \\
\quad (\forall \Pi. \\
\qquad \textsf{MEM}\ \Pi\ \Pi_l \Rightarrow \\
\qquad\quad \textsf{valid-prob}\ \Pi \wedge \Pi.I \sqsubseteq x \wedge \Pi\ \textsf{solved-by}\ \mathit{solve}\ \Pi \wedge \\
\qquad\quad \textsf{sat-pre}\ (\mathcal{N}(\Pi), \mathit{solve}\ \Pi)) \Rightarrow \\
\qquad (\textbf{let} \\
\qquad\quad \mathit{sub_prob_plans} = \textsf{MAP}\ \mathit{solve}\ \Pi_l\ ; \\
\qquad\quad \mathit{concatenated_plans} = \textsf{FLAT}\ \mathit{sub_prob_plans} \\
\qquad\ \textbf{in} \\
\qquad\quad \forall \Pi.\ \textsf{MEM}\ \Pi\ \Pi_l \Rightarrow \Pi.G \sqsubseteq \mathsf{ex}(x, \mathit{concatenated_plans}))
\end{array}$$

In the HOL4 statement above (i) $\lhd_l$ is a predicate that lifts precedence to lists of problems, where for a list of problems $\Pi_1 \ldots \Pi_N$, it denotes that $\Pi_j \lhd \Pi_k$ holds, for all $j < k \leq N$, and (ii) *solve* is a function that maps every planning problem to a plan that solves it.

Before we discuss the proof of this lemma, we define the following union operation on planning problems and a lifted union operation for lists of planning problems.

▷ Definition 7 (Planning Problem Union).

$$\Pi_1 \cup \Pi_2 \stackrel{\text{def}}{=} \\ <|\, I := \Pi_1.I \uplus \Pi_2.I;\ \delta := \Pi_1.\delta \cup \Pi_2.\delta;\ G := \Pi_1.G \uplus \Pi_2.G \,|>$$

$$\bigcup \Pi_l \stackrel{\text{def}}{=} \mathsf{FOLDR} \cup \Pi_\emptyset\ \Pi_l$$

$\Pi_\emptyset$ is the "empty problem", whose initial and goal states have an empty domain, i.e. states mapping nothing to nothing, and that does not have actions.

The following theorem shows that the semantics of the planning problem union operations are as intended.

$$\vdash (\forall \Pi.\ \mathsf{MEM}\ \Pi\ \Pi_l \Rightarrow \mathsf{valid\text{-}prob}\ \Pi) \Rightarrow \mathsf{valid\text{-}prob}\ (\bigcup \Pi_l)$$

Informally, a sketch of the proof of Lemma 1 follows.

Proof. The proof is by induction on the list Π_l. The base case is trivial. In the step case we have the theorem for list of problems Π_l, and we need to show that it applies to Π_l with the problem Π pre-pended to it. The key idea of the proof is to deal with $\bigcup \Pi_l$ as one planning problem. Since Π precedes every problem in Π_l, we have that Π precedes $\bigcup \Pi_l$. From this, the inductive hypothesis, Proposition 1, Proposition 2, and Proposition 3, the result follows. ◀

Before this verification, we missed the condition $\mathsf{sat\text{-}pre}\ (\mathcal{N}(\Pi), f\ \Pi)$ from the assumptions of Lemma 1. This condition forbids plans with actions whose preconditions are unsatisfied during isolated execution in the corresponding problem – i.e. such actions are ignored by the execution function when considering the problem in isolation. The importance of this condition shall be discussed in detail in Section 6.

4 Covering via Concatenation

Having just developed conditions for plan synthesis via concatenation, it remains to understand how a concrete problem may be broken up into an ordered list of sub-problems, so that a concatenation of sub-problems plans corresponds to a plan for the concrete problem. First, this will require that we formally treat the question of what it is to be a sub-problem. We then establish a concept of coverage, so that when a concrete problem is *covered* by a list of sub-problems, we have the core sufficient condition to concatenate sub-problem plans according to a schema analogous to Lemma 1.

A problem is a sub-problem of another, if the constituents—states and actions—of the former are subsets/submaps of corresponding constituents of the latter.

▷ Definition 8 (Sub-problem). Problem Π_1 is a sub-problem of Π_2, written $\Pi_1 \subseteq \Pi_2$, if $I_1 \subseteq I_2$, and if $\delta_1 \subseteq \delta_2$.

$$\Pi_1 \subseteq \Pi_2 \stackrel{\text{def}}{=} \Pi_1.I \sqsubseteq \Pi_2.I \wedge \Pi_1.\delta \subseteq \Pi_2.\delta$$

▷ Definition 9 (Covering Problems). A list of planing problems Π_l covers a problem Π iff (i) every member of Π_l is a sub-problem of Π and (ii) every goal of Π is a goal for some member of Π_l.

$$\begin{aligned}&\mathsf{covers}\ \Pi_l\ \Pi \stackrel{\text{def}}{=} \\ &\quad (\forall x. \\ &\qquad x \in \mathcal{D}(\Pi.G) \Rightarrow \\ &\qquad\quad \exists \Pi'.\ \mathsf{MEM}\ \Pi'\ \Pi_l \wedge x \in \mathcal{D}(\Pi'.G) \wedge \Pi.G\ `\ x = \Pi'.G\ `\ x) \wedge \\ &\quad \forall \Pi'.\ \mathsf{MEM}\ \Pi'\ \Pi_l \Rightarrow \Pi' \subseteq \Pi\end{aligned}$$

▷ Example 6. Let the problem Π_1'' be s.t. $\Pi_1''.I = \{v_3, v_1, v_4\}$, $\Pi_1''.\delta = \{(\{v_1, v_3\}, \{\overline{v_3}, \overline{v_4}\}), (\emptyset, \{v_3\})\}$, and $\Pi_1''.G = \{\overline{v_4}\}$. Let the problem Π_1''' be s.t. $\Pi_1'''.I = \{v_3, v_2, v_5\}$, $\Pi_1'''.\delta = \{(\{v_2, v_3\}, \{\overline{v_3}, \overline{v_5}\}), (\emptyset, \{v_3\})\}$, and $\Pi_1'''.G = \{\overline{v_5}\}$. The list $[\Pi_1''; \Pi_1''']$ covers the problem Π_1 since $\Pi_1'' \subseteq \Pi_1$ and $\Pi_1''' \subseteq \Pi_1$, and since Π_1'' covers the goal $\overline{v_4}$ in Π_1, Π_1''' covers the goal $\overline{v_5}$ in Π_1.

We now establish sufficient conditions for the concatenation of sub-problem plans to solve the corresponding concrete problem. This result is a consequence of Lemma 1.

▷ Theorem 1. Consider a set $\Pi_1 \ldots \Pi_N$ of problems that covers Π, satisfying $\Pi_j \lhd \Pi_k$ for all $j < k \leq N$. For $1 \leq i \leq N$ let $\vec{\pi}_i$ be a plan for Π_i. Then rem-cless$(\mathcal{N}(\Pi_1), \vec{\pi}_1)$ +rem-cless$(\mathcal{N}(\Pi_2), \vec{\pi}_2)$ + ... rem-cless$(\mathcal{N}(\Pi_N), \vec{\pi}_N)$ is a plan for Π.

```
⊢ covers Π_l Π ∧ ◁_l Π_l ⇒
    (∀Π'. MEM Π' Π_l ⇒ valid-prob Π' ∧ Π' solved-by f Π') ⇒
      (let
          inst_plans = MAP (λΠ'. rem-cless (N(Π'),[],f Π')) Π_l ;
          concatenated_plans = FLAT inst_plans
       in
          Π solved-by concatenated_plans)
```

Note that in the theorem above, the sub-problem plans can be concatenated to solve the concrete problem after removing actions with unsatisfied preconditions. Such actions are removed by the function rem-cless. This is required to ensure that the assumption sat-pre satisfied, as is required for every sub-problem in Lemma 1. This function was not in our originally published algorithm, and is defined as follows:

```
rem-cless (x,pfx,(p,e)::π⃗) ≝
  if p ⊑ ex(x,pfx) then rem-cless (x,pfx + [(p,e)],π⃗)
  else rem-cless (x,pfx,π⃗)
rem-cless (x,pfx,[]) ≝ pfx
```

The following two theorems show that rem-cless: (i) provides a list of actions whose preconditions are always satisfied during execution, and (ii) does not effect the results of execution in isolation in a sub-problem.

```
⊢ sat-pre (x,rem-cless (x,[],π⃗))
```

```
⊢ ex(x,π⃗) = ex(x,rem-cless (x,[],π⃗))
```

During formalisation work related to Theorem 1, we discovered an error in our original conception of the definition of what a sub-problem is. Before this verification, we omitted the requirement that $\Pi_1.I \sqsubseteq \Pi_2.I$, opting for the erroneous condition $\mathcal{D}(\Pi_1) \subseteq \mathcal{D}(\Pi_2)$. This faulty definition allows for sub-problems of the same problem to have conflicting initial states, in which case the assumption of having more than one sub-problem becomes an insufficient assumption to prove the algorithm's soundness.

5 Concatenating Instantiations of a Quotient Plan

Theorem 1 establishes sufficient conditions enabling the synthesis of a concrete plan by concatenating plans for sub-problems. In fact, our compositional approach allows an additional efficiency: since it treats the scenario where each sub-problem is isomorphic, only one plan

need ever be computed. That one plan is then *instantiated* for a covering set of isomorphic sub-problems. Finally, a concrete plan is synthesised by concatenating the instantiated sub-problem plans. This algorithm requires a canonical sub-problem, the quotient problem, which is isomorphic to each sub-problem of the concrete problem at hand. To permit sub-plan concatenation, successive sub-problems must satisfy the sub-problem precedence relation. To ensure this, our algorithm augments the quotient, ensuring that shared resources are left as they are found between sub-plan executions.

5.1 Formalising Instantiations

An instantiation maps constituents from a planning problem Π_1 to those from another problem Π_2, by mapping the state variables that underlie the mapped constituent. In particular, it is a function that explicitly maps the quotient into a sub-problem of the concrete problem by mapping: (i) quotient state variables to state variables of the concrete problem, (ii) quotient states to concrete problem states, and (iii) the quotient's actions to concrete problem actions.

To formulate that in HOL4, for state variables, instantiation is a function from $\mathcal{D}(\Pi_1)$ to $\mathcal{D}(\Pi_2)$. For states, it was a surprising challenge to define in HOL4 what an instantiation is. Because states are finite maps, the instantiation $\pitchfork(\!|x|\!)$ of a state x is an application of an image of the instantiation $\pitchfork$ to the domain of the state. Instantiation here is therefore described as a function image application. For example, for a state $\{o_1 \mapsto T, o_2 \mapsto F\}$ and an instantiation function $\pitchfork$, the instantiation of that state using that function is the state $\{\pitchfork(o_1) \mapsto T, \pitchfork(o_2) \mapsto F\}$. This is equivalent to composing the inverse of the instantiation function with the state.

▷ Definition 10 (State Instantiation). Instantiation of state x with instantiation $\pitchfork$ is defined as the composition of x with the inverse of $\pitchfork$:

$$\pitchfork(\!|x|\!) \stackrel{\text{def}}{=} x \circ \pitchfork^{-1}$$

Overloading the $(\!|\,|\!)$ notation, below we define the instantiation operation, for (i) an action, (ii) a factored system, (iii) a planning problem, and (iv) an action sequence, respectively.

$$\pitchfork(\!|(p,e)|\!) \stackrel{\text{def}}{=} (\pitchfork(\!|p|\!),\pitchfork(\!|e|\!))$$

$$\pitchfork(\!|\delta|\!) \stackrel{\text{def}}{=} (\lambda\,\pi.\ \pitchfork(\!|\pi|\!))(\!|\delta|\!)$$

$$\pitchfork(\!|\Pi|\!) \stackrel{\text{def}}{=} \Pi\ \texttt{with <|}\ I\ \texttt{:=}\ \pitchfork(\!|\Pi.I|\!)\texttt{;}\ \delta\ \texttt{:=}\ \pitchfork(\!|\Pi.\delta|\!)\texttt{;}\ G\ \texttt{:=}\ \pitchfork(\!|\Pi.G|\!)\ \texttt{|>}$$

$$\pitchfork(\!|\vec{\pi}|\!) \stackrel{\text{def}}{=} \mathsf{MAP}\ (\lambda\,\pi\texttt{.}\ \pitchfork(\!|\pi|\!))\ \vec{\pi}$$

▷ Example 7. Recall from Example 3 the quotient Π'_1 of the concrete problem Π_1. Let instantiation $\pitchfork$ be $\{p_1 \mapsto v_1, p_2 \mapsto v_3, p_3 \mapsto v_4\}$. The problem Π''_1 from Example 6 is the same as $\pitchfork(\!|\Pi'_1|\!)$, i.e. it is the instantiation of Π'_1 using $\pitchfork$.

Let valid-inst $\pitchfork$ mean that $\pitchfork$ is a bijection. We have the following theorems.

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \Rightarrow\ \mathsf{ex}(\pitchfork(\!|x|\!),\pitchfork(\!|\vec{\pi}|\!))\ =\ \pitchfork(\!|\mathsf{ex}(x,\vec{\pi})|\!)$$

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \wedge\ \Pi\ \mathsf{solved\text{-}by}\ \vec{\pi}\ \Rightarrow\ \pitchfork(\!|\Pi|\!)\ \mathsf{solved\text{-}by}\ \pitchfork(\!|\vec{\pi}|\!)$$

$$\vdash \mathsf{valid\text{-}inst}\ \pitchfork\ \Rightarrow\ \mathcal{D}(\mathcal{N}(\pitchfork(\!|\Pi|\!)))\ =\ \pitchfork(\!|\mathcal{D}(\mathcal{N}(\Pi))|\!)$$

Note that, in our original treatment [5], we did not explicitly state bijectivity of instantiations as a condition. This is because it is a consequence of the fact that the instantiations we considered then were transversals of equivalence classes of state variables under symmetry—a.k.a. *orbits*. Orbits form a partition of the domain of a planning problem, and since a transversal maps every orbit to one of its members, transversals are bijective.

Our algorithm also requires the following additional condition on sets of instantiations:

▷ Definition 11 (Valid Set of Instantiations). Any two different instantiations from a set of instantiations Δ should not map different variables from the domain of the quotient to the same variable in their range.

$$\begin{array}{l}\textsf{pwise-valid}\ \ \Delta\ \ vs \stackrel{\text{def}}{=} \\ \quad \forall \pitchfork_1\ \pitchfork_2\ \ v_1\ \ v_2\ .\ \\ \qquad \pitchfork_1\ \in\ \Delta\ \wedge\ \pitchfork_2\ \in\ \Delta\ \wedge\ v_1\ \in\ vs\ \wedge\ v_2\ \in\ vs\ \wedge\ v_1\ \neq\ v_2\ \Rightarrow \\ \qquad\quad \pitchfork_1\ \ v_1\ \neq\ \pitchfork_2\ \ v_2\end{array}$$

This condition guarantees that different instantiations of the same state are consistent with each other—i.e., a distinct variable is mapped to the same value in all the instantiated states. Again, we did not state this assumption explicitly in our original treatment since it holds for instantiations that are transversals of the state variable orbits, because a set of orbits forms a partition of the domain of the planning problem.

5.2 Planning via an Augmented Quotient

We now establish the correctness of the the target algorithm, which synthesises a concrete plan by concatenating a set of covering instantiations of the solution to an augmented quotient. The algorithm inputs are (i) a quotient of the concrete problem, (ii) a solution to that quotient, and (iii) a set of instantiations of the quotient which cover the concrete problem. In order to leverage the previous results in formally verifying that the target algorithm is correct, the key remaining task is to deal with that, as yet, we have no ordering of instantiations of isomorphic sub-problems. Theorem 1 is not directly applicable without a notion of precedence. We shall establish below the conditions so that any two instantiations of the augmented quotient can participate in the precedence relation together. Therefore, any ordering of such instantiations is admissible for the purposes of leveraging the algorithm verified in Theorem 1.

Two necessary concepts to state conditions guaranteeing that instantiations can participate in the precedence relation are *common variables* and *sustainable variables*. Given a set of instantiations, the common variables are variables mapped to the same value by at least two instantiations. A sustainable variable holds the same assignment in the initial state as it does in a goal state. Formally, they are defined as follows:

▷ Definition 12 (Common Variables). For a set of instantiations Δ, the set of *common variables*, written $\bigcap_v \Delta\ vs$, comprises all elements from the given set of variables vs that occur in the ranges of more-than-one member of Δ.

$$\begin{array}{l}\bigcap_v\ \ \Delta\ \ vs \stackrel{\text{def}}{=} \\ \quad \{\, v\ \mid\ \exists \pitchfork_1\ \pitchfork_2\ .\ \pitchfork_1\ \in\ \Delta\ \wedge\ \pitchfork_2\ \in\ \Delta\ \wedge\ \pitchfork_1\ \neq\ \pitchfork_2\ \wedge\ v\ \in\ vs\ \wedge\ \pitchfork_1\ \ v\ =\ \pitchfork_2\ \ v\,\}\end{array}$$

▷ Example 8. Let instantiation $\pitchfork'$ be $\{p_1 \mapsto v_2, p_2 \mapsto v_3, p_3 \mapsto v_5\}$. Take Δ to be $\{\pitchfork, \pitchfork'\}$. For Δ, we have $\bigcap_v \Delta\ \{p_1, p_2, p_3\} = \{p_1\}$.

▷ Definition 13 (Sustainable Variables). A set of variables vs is sustainable in a problem Π iff $I\lfloor_{vs} = G\lfloor_{vs}$.

$\mathsf{sustainable}\ \Pi\ vs \stackrel{\text{def}}{=} \Pi.I\downarrow_{vs} = \Pi.G\downarrow_{vs}$

Our headline result relies on the following argument. If the intersection of—the needed variables of the quotient problem, with the common variables from instantiations Δ—are sustained in Π, then any pair of distinct instantiations of Π are elements in the precedence relation.

$$\begin{aligned}\vdash\ & \mathsf{valid\text{-}inst}\ \pitchfork_1 \wedge \mathsf{valid\text{-}inst}\ \pitchfork_2 \wedge \\ & \mathsf{pwise\text{-}valid}\ \Delta\ \mathcal{D}(\Pi) \wedge \mathsf{valid\text{-}prob}\ \Pi \wedge \pitchfork_1 \in \Delta \wedge \\ & \pitchfork_2 \in \Delta \wedge \pitchfork_1 \neq \pitchfork_2 \wedge \mathsf{sustainable}\ \Pi\ (\textstyle\bigcap_v \Delta\ \mathcal{D}(\Pi) \cap \mathcal{D}(\mathcal{N}(\Pi))) \Rightarrow \\ & \quad \pitchfork_1(\!|\Pi|\!) \lhd \pitchfork_2(\!|\Pi|\!)\end{aligned}$$

From this and from Theorem 1 we derive our headline theorem, stating soundness conditions for planning via a quotient. In this theorem we refer to a planning problem, Π', as being a descriptive quotient of some other problem Π. The stated conditions of the theorem define how some problem Π' qualifies as a descriptive quotient of Π.

▷ **Theorem 2.** Consider a problem Π, a descriptive quotient Π', a solution $\vec{\pi}'$ to Π', and a set of instantiations Δ. Suppose $\{\pitchfork(\!|\Pi'|\!) \mid \pitchfork \in \Delta\} (= \mathbf{\Pi})$ covers Π, and $\bigcap_v \Delta\ \mathcal{D}(\Pi'_1) \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$ are sustainable in Π'. Then any concatenation of the plans $\{\mathsf{rem\text{-}class}(\mathcal{N}(\pitchfork(\!|\Pi'|\!)), [], \pitchfork(\!|\vec{\pi}'|\!)) \mid \pitchfork \in \Delta\}$ solves Π.

Note that the theorem above requires, for a quotient, that the intersection of its needed variables with the common variables between instantiations are sustainable. If this requirement is not satisfied by a quotient, the concatenated quotient plan instantiations might not solve the concrete problem, as shown below.

▷ **Example 9.** Note that $\mathcal{D}(\mathcal{N}(\Pi'_1)) = \{p_1, p_2\}$, and recall that $\bigcap_v \Delta\ \{p_1, p_2, p_3\} = \{p_1\}$. Thus the intersection of the quotient's needed variables with the common variables between instantiations, $\bigcap_v \Delta\ \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$, is $\{p_1\}$. The quotient Π'_1 does not sustain that intersection since the assignment of p_1 in $\Pi'_1.I$ does not occur in the goal $\Pi'_1.G$. Now, to see the problem this might cause, we instantiate the descriptive quotient Π'_1 with $\pitchfork'$, which yields problem Π'''_1 from Example 6. Thus $[\pitchfork(\!|\Pi'_1|\!); \pitchfork'(\!|\Pi'_1|\!)]$ covers the problem Π_1. A plan for the descriptive quotient Π'_1 is $\vec{\pi}' \equiv [(\{p_1, p_2\}, \{\overline{p}_3, \overline{p}_1\})]$ and its two instantiations are $\pitchfork(\!|\vec{\pi}'|\!) = [\pi_2]$ and $\pitchfork'(\!|\vec{\pi}'|\!) = [\pi_3]$. However, the two possible concatenations of $\pitchfork(\!|\vec{\pi}'|\!)$ and $\pitchfork'(\!|\vec{\pi}'|\!)$ do not solve Π_1 because both plans, $\pitchfork(\!|\vec{\pi}'|\!)$ and $\pitchfork'(\!|\vec{\pi}'|\!)$, require v_3 initially, but do not establish it.

To guarantee that the intersection of the quotient's needed variables with the instantiations' common variables are sustainable, the goal of a quotient Π' is augmented with assignments that guarantee that the quotient sustains those variables. In particular, the quotient's goal should be augmented by the assignment of the variables $\bigcap_v \Delta\ \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$ in the quotient's initial state. This step should be performed before the quotient is solved. The next example gives a concrete example of this augmentation.

▷ **Example 10.** To solve Π_1 via solving Π'_1, we augment the goal $\Pi'_1.G$ with the initial state assignment of the variables in $\bigcap_v \Delta\ \{p_1, p_2, p_3\} \cap \mathcal{D}(\mathcal{N}(\Pi'_1))$, i.e. $\Pi'_1.I\downarrow_{\bigcap_v \Delta\ \{p_1,p_2,p_3\} \cap \mathcal{D}(\mathcal{N}(\Pi'_1))}$. The resulting problem, Π^q_1, is equal to Π'_1 except that it has the literal $\{p_1\}$ added to its goals, so $\Pi^q_1.G = \{p_1, \overline{p}_3\}$. A plan for Π^q_1 is $\vec{\pi}^q \equiv [(\{p_1, p_2\}, \{\overline{p}_3, \overline{p}_1\}); (\emptyset, \{p_1\})]$, and two instantiations of it are $\pitchfork(\!|\vec{\pi}^q|\!) = [\pi_2; \pi_1]$ and $\pitchfork'(\!|\vec{\pi}^q|\!) = [\pi_3; \pi_1]$. Concatenating $\pitchfork(\!|\vec{\pi}^q|\!)$ and $\pitchfork'(\!|\vec{\pi}^q|\!)$ in any order solves Π_1.

The fact that goal augmentation works is shown in the following theorem.

$$\vdash \textbf{let}\ \Pi^q = \Pi'\ \texttt{with}\ G\ \texttt{:=}\ \Pi'.I\restriction_{\bigcap_v\ (\text{set}\ \Delta)\ \mathcal{D}(\Pi')\ \cap\ \mathcal{D}(\mathcal{N}(\Pi'))}\ \uplus\ \Pi'.G\ \textbf{in}\ \text{sustainable}\ \Pi^q\ (\textstyle\bigcap_v\ (\text{set}\ \Delta)\ \mathcal{D}(\Pi^q)\ \cap\ \mathcal{D}(\mathcal{N}(\Pi^q)))$$

Our headline result now follows straightforwardly from the manner in which the quotient augmentation operates. Because a quotient with an augmented goal sustains the common needed variables, the algorithm from Theorem 2 can be used to synthesise a concrete problem solution by instantiating and concatenating the augmented quotient solutions.

$$\begin{aligned}\vdash\ &\textbf{let}\\ &\quad \Pi^q = \Pi'\ \texttt{with}\ G\ \texttt{:=}\ \Pi'.I\restriction_{\bigcap_v\ (\text{set}\ \Delta)\ \mathcal{D}(\Pi')\ \cap\ \mathcal{D}(\mathcal{N}(\Pi'))}\ \uplus\ \Pi'.G\ \texttt{;}\\ &\quad \mathit{inst\_plans} = \text{MAP}\ (\lambda \pitchfork.\ \text{rem-class}\ (\mathcal{N}(\pitchfork(\!|\Pi^q|\!)), \texttt{[]}, \pitchfork(\!|\overrightarrow{\pi}^q|\!)))\ \Delta\ \texttt{;}\\ &\quad \mathit{concatenated\_plans} = \text{FLAT}\ \mathit{inst\_plans}\\ &\textbf{in}\\ &\quad \text{ALL-DISTINCT}\ \Delta \wedge (\forall \pitchfork.\ \text{MEM}\ \pitchfork\ \Delta \Rightarrow \text{valid-inst}\ \pitchfork) \wedge\\ &\quad \text{valid-prob}\ \Pi' \wedge\\ &\quad \text{INJ}\ (\lambda \pitchfork.\ \pitchfork(\!|\Pi^q|\!))\ (\text{set}\ \Delta)\ \mathcal{U}(\texttt{:}(\alpha, \beta)\ \mathit{planningProblem}) \wedge\\ &\quad \text{pwise-valid}\ (\text{set}\ \Delta)\ \mathcal{D}(\Pi^q) \wedge \text{covers}\ (\text{MAP}\ (\lambda \pitchfork.\ \pitchfork(\!|\Pi^q|\!))\ \Delta)\ \Pi \wedge\\ &\quad \Pi^q\ \text{solved-by}\ \overrightarrow{\pi}^q \Rightarrow\\ &\qquad \Pi\ \text{solved-by}\ \mathit{concatenated\_plans}\end{aligned}$$

In closing, it is worth noting that Theorem 2 assumes nothing in the way the augmented quotient is computed. We believe this in itself is an important extension to our earlier work [5], which was limited to situations where the quotient under consideration is computed according to identified symmetric variables — i.e. as per Π' in Example 3. Our new results describe an algorithm that is applicable to descriptive quotients computed in problems that may not have symmetries. It is applicable provided the descriptive quotient is isomorphic to a set of sub-problems covering the concrete problem.

Also, the planning problem is of type (α, β) *planningProblem*. This denotes that indeed the algorithm for composing solutions is applicable to planning problems whose state variables can be assigned to values of any type β, without any constraints on that type. Additionally the cardinality of the set of actions in the problem or the quotient is unconstrained. Thus the planning problem and its quotient are not necessarily propositionally factored systems, making planning via descriptive quotients applicable to planning problems with infinite states, like numeric planning.

Lastly we note that the new algorithm, which includes a call to the function rem-cless, suffers almost no run-time penalty compared to the original algorithm which did not include a call to rem-cless. This is because the run-time of rem-cless is linear in the length of the quotient plan, whose length in most benchmarks is linear in the problem size. Indeed, the overall run-time is dominated by finding a plan for the quotient.

6 Fixing the Algorithm via Formalisation

One benefit of our work is the discovery and correction of an easy-to-miss bug in our original algorithm [5]. We now describe that bug, first intuitively and then using a detailed example. Suppose a plan is found for a quotient system, and that plan contains a *spurious* action: an action whose precondition is not satisfied when that action is scheduled to execute—i.e., we

have not applied the rem-cless function. Now consider the case that the plan is instantiated multiple times, and the results of this are concatenated together to form a concrete plan. When we execute the first instantiation of the quotient's plan, no error occurs. However, that execution may have a "side effect", so that later instantiations of the spurious action now have an effect. It can be the case that such a spurious effect interferes with the plan execution, rendering the concrete plan invalid as follows.

For notational economy, let action schemata π_1, π_2, π_3, and π_4 be defined as $\pi_1(x, y, z) \equiv (\{x, y\}, \{\overline{y}, \overline{z}\})$, $\pi_2(x) \equiv (\emptyset, \{x\})$, $\pi_3(x) \equiv (\emptyset, \{\overline{x}\})$, and $\pi_4(x, y) \equiv (\{\overline{x}\}, \{y\})$, respectively. Consider a planning problem Π where $\Pi.I \equiv \{v_1, v_2, v_3, v_4, v_5, v_6, \overline{v_7}\}$, $\Pi.\delta \equiv \{\pi_1(v_1, v_3, v_4), \pi_1(v_2, v_3, v_5), \pi_2(v_3), \pi_4(v_6, v_7), \pi_3(v_6)\}$, and $\Pi.G \equiv \{\overline{v_4}, \overline{v_5}, \overline{v_6}, \overline{v_7}\}$. Also consider Π', a quotient of Π, where $\Pi'.I \equiv \{p_1, p_2, p_3, p_4, \overline{p_5}\}$, $\Pi'.\delta \equiv \{\pi_1(p_1, p_2, p_3), \pi_2(p_2), \pi_4(p_4, p_5), \pi_3(p_4)\}$, and $\Pi'.G \equiv \{\overline{p_3}, \overline{p_4}, \overline{p_5}\}$. Consider the two instantiations $\pitchfork$ and $\pitchfork'$ defined as $\pitchfork \equiv \{p_1 \mapsto v_1, p_2 \mapsto v_3, p_3 \mapsto v_4, p_4 \mapsto v_6, p_5 \mapsto v_7\}$, and $\pitchfork' \equiv \{p_1 \mapsto v_2, p_2 \mapsto v_3, p_3 \mapsto v_5, p_4 \mapsto v_6, p_5 \mapsto v_7\}$. Let the set of instantiations Δ be $\{\pitchfork, \pitchfork'\}$. The problem Π is covered by $\pitchfork(\!|\Pi'|\!)$ and $\pitchfork'(\!|\Pi'|\!)$, since they are sub-problems of Π and they cover its goal $\Pi.G$. The first step of the algorithm would be to augment the quotient's goal with $\Pi'.I{\downarrow}_{\bigcap_v \Delta\, \mathcal{D}(\Pi') \cap \mathcal{D}(\mathcal{N}(\Pi'))}$. We have $\bigcap_v \Delta\, \mathcal{D}(\Pi') = \{p_1, p_4, p_5\}$, i.e. there are three common variables between $\pitchfork$ and $\pitchfork'$. Also, the needed variables of the quotient are $\mathcal{D}(\mathcal{N}(\Pi')) = \{p_1, p_2\}$, since both p_1 and p_2 occur with the same assignments in the quotient's action preconditions and its initial state. Thus the goal of Π' is augmented with the literal $\{p_1\}$ resulting in the problem Π^q which is the same as Π' except that its goal $\Pi^q.G$ is $\{p_1, \overline{p_3}, \overline{p_4}, \overline{p_5}\}$. Next, the algorithm searches for a plan for Π^q. One such plan is $\overrightarrow{\pi}^q \equiv [\pi_1(p_1, p_2, p_3); \pi_2(p_1); \pi_4(p_4, p_5); \pi_3(p_4)]$. Note: when $\overrightarrow{\pi}^q$ is executed at the initial state $\Pi^q.I$, the action $\pi_4(p_4, p_5)$ will have no effect since its precondition, $\overline{p_4}$, will not hold before when it executes.

The next step is computing instantiations of $\overrightarrow{\pi}^q$, which are $\pitchfork(\!|\overrightarrow{\pi}^q|\!) = [\pi_1(v_1, v_3, v_4); \pi_2(v_3); \pi_4(v_6, v_7); \pi_3(v_6)]$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!) = [\pi_1(v_2, v_3, v_5); \pi_2(v_3); \pi_4(v_6, v_7); \pi_3(v_6)]$. Then the algorithm returns the concatenation of $\pitchfork(\!|\overrightarrow{\pi}^q|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!)$ in any order as a solution to Π. However, any concatenation of $\pitchfork(\!|\overrightarrow{\pi}^q|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!)$ does *not* solve Π since the last occurrence of $\pi_4(v_6, v_7)$ in the concatenation will execute successfully. This is because the first occurrence of $\pi_3(v_6)$ sets the precondition of $\pi_4(v_6, v_7)$, and the execution of $\pi_4(v_6, v_7)$ will set v_7 to true, which contradicts the goal of Π.

The verified algorithm, however, returns a concatenation of the action sequences rem-cless $(\mathcal{N}(\pitchfork(\!|\Pi^q|\!)), [\,], \pitchfork(\!|\overrightarrow{\pi}^q|\!))$ and rem-cless $(\mathcal{N}(\pitchfork'(\!|\Pi^q|\!)), [\,], \pitchfork'(\!|\overrightarrow{\pi}^q|\!))$. This is a solution for Π since rem-cless removes $\pi_4(v_6, v_7)$ from both $\pitchfork(\!|\overrightarrow{\pi}^q|\!)$ and $\pitchfork'(\!|\overrightarrow{\pi}^q|\!)$ as its preconditions are not met.

Interestingly, the possible bad scenario never showed up in any of the thousands of standard planning benchmarks on which we conducted our earlier experiments. We were lucky that the planner we used never produced plans with spurious actions. Nonetheless, we cannot afford to leave possible bugs latent in such corner cases if AI algorithms are to be deployed in a safety sensitive applications. Needless to say, our discovery of this bug further strengthens the argument for using formal verification for AI algorithms.

7 Related Work

The compositional approach to AI planning is very effective. A prominent example is planning using abstractions based on projection, exploiting acyclicity in variable dependencies [25, 39]. Also *factored planning* abstracts a problem into multiple "factors", which are obtained using a tree decomposition of a graph representation of variable dependencies [7, 11, 24].

Despite that extensive literature, to our knowledge, this is the first verification of a

compositional planning algorithm. Indeed, most applications of formal methods to the area of AI planning were in the context of reasoning about planning domain models and plans and verifying properties of them, and not verifying planning algorithms. For instance, model checkers were used to validate that classical planning domain models satisfy given specifications [29, 21, 36]. Also, model checkers were used to verify safety and temporal properties of plans [19, 18]. Similar applications of model checking also exist for other planning formalisms, such as temporal planning [9]. Since the only formal technique used by earlier work was model checking, the limitation to only verifying model and plan properties, versus verifying planning algorithms, should come as no surprise. This is due to the limitations on what can be represented in model checkers and their formalisms.

The only application of theorem provers based formal methods to planning was by Abdulaziz and Lammich [4], who developed and formally verified a tool to validate planning domain models and plans. Other verification work using theorem provers relevant to our work is the verification of model checking algorithms, where the motivation is to obtain formally verified model checking algorithms and implementations [37, 32, 16, 12]. However, we note that although those model checking algorithms use abstractions based techniques, like partial order reduction, they are not compositional algorithms. Other related work is on formalising automata theory. For instance, textbook results in automata theory have been formalised on a number of occasions and in different logics [14, 15, 28, 40].

8 Conclusion

We verified a compositional AI planning algorithm that we published earlier, and found mistakes in its pen-and-paper formulation. This is similar to our earlier experience [1, 2, 3, 6], when we found mistakes in our and other people's work. We believe that planning may be particularly prone to such errors due to its heavy combinatorial nature, making it easy to miss corner cases, as well as the dense usage of notation in the planning literature. Although such errors can be corner cases, they cannot be tolerated in safety-critical applications such as outer space exploration, making a strong case for the utility of mechanical verification.

Furthermore, formalising the algorithm in a theorem prover made it easier to generalise our algorithm from planning problems with propositional state variables to problems in which state variables are not necessarily Boolean, finite or even countable. This raises the possibility of applying this algorithm to temporal planning, numeric planning, or hybrid planning. However, this might need extending the existing theory to reason about actions whose preconditions and effects are functions in state variables, versus assignments to state variables. Since we do not assume that the planning problem has a finite number of actions, we hypothesise that a lot of the theory developed here could be reused for richer planning formalisms by showing that planning problems from those formalisms could be reduced to problems represented in our theory.

We made a number of observations in our efforts which we believe provide insight into how HOL4 can be improved. A feature of HOL4 which we would cite as the most positive is the ease of modifying or adding tactics, since the entire system is completely implemented in SML. Also, automation tactics in general are reasonable, and surprisingly proved some lemmas completely automatically, modulo providing the methods with the appropriate lists of theorems. Two high-level issues we encountered were difficulties in searching for theorems (something we believe all systems struggle with) and the need to repeat theorem-hypotheses from goal to goal. This latter issue would be much-ameliorated by a mechanism akin to Isabelle's locales or Coq's sections.

References

1. Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. Mechanising Theoretical Upper Bounds in Planning. In *Workshop on Knowledge Engineering for Planning and Scheduling*, 2014.
2. Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. Verified Over-Approximation of the Diameter of Propositionally Factored Transition Systems. In *Interactive Theorem Proving*, pages 1–16. Springer, 2015.
3. Mohammad Abdulaziz, Charles Gretton, and Michael Norrish. A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2017.
4. Mohammad Abdulaziz and Peter Lammich. A formally verified validator for classical planning problems and solutions. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 474–479. IEEE, 2018.
5. Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th International Joint Conference on Artificial Intelligence, IJCAI*, pages 25–31, 2015.
6. Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Formally Verified Algorithms for Upper Bounding State Space Diameters. In *To appear in the Journal of Automated Reasoning*, 2017.
7. Eyal Amir and Barbara Engelhardt. Factored planning. In *IJCAI*, volume 3, pages 929–935. Citeseer, 2003.
8. Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 6094–6101, 2018.
9. Saddek Bensalem, Klaus Havelund, and Andrea Orlandini. Verification and validation meet planning and scheduling, 2014.
10. Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49 – 107, 2000.
11. Ronen I Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *AAAI*, volume 6, pages 809–814, 2006.
12. Julian Brunner and Peter Lammich. Formal verification of an executable ltl model checker with partial order reduction. In *NASA Formal Methods Symposium*, pages 307–321. Springer, 2016.
13. Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.
14. Robert L Constable, Paul B Jackson, Pavel Naumov, and Juan C Uribe. Constructively formalizing automata theory. In *Proof, language, and interaction*, pages 213–238, 2000.
15. Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. A constructive theory of regular languages in Coq. In *International Conference on Certified Programs and Proofs*, pages 82–97. Springer, 2013.
16. Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *International Conference on Computer Aided Verification*, pages 463–478. Springer, 2013.
17. Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.
18. Maria Fox, Richard Howey, and Derek Long. Exploration of the robustness of plans. In *AAAI*, pages 834–839, 2006.

19 Robert P Goldman, Ugur Kuter, and Tony Schneider. Using classical planners for plan verification and counterexample generation. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

20 Marc Hanheide, Moritz Göbelbecker, Graham S. Horn, Andrzej Pronobis, Kristoffer Sjöö, Alper Aydemir, Patric Jensfelt, Charles Gretton, Richard Dearden, Miroslav Janícek, Hendrik Zender, Geert-Jan M. Kruijff, Nick Hawes, and Jeremy L. Wyatt. Robot task planning and explanation in open and uncertain worlds. *Artif. Intell.*, 247:119–150, 2017.

21 Klaus Havelund, Alex Groce, Gerard Holzmann, Rajeev Joshi, and Margaret Smith. Automated testing of planning models. In *International Workshop on Model Checking and Artificial Intelligence*, pages 90–105. Springer, 2008.

22 C Norris Ip and David L Dill. Verifying systems with replicated components in murϕ. In *International Conference on Computer Aided Verification*, pages 147–158. Springer, 1996.

23 C Norris Ip and David L Dill. Verifying systems with replicated components in murϕ. *Formal Methods in System Design*, 14(3):273–310, 1999.

24 Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In *IJCAI*, pages 1942–1947, 2007.

25 C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

26 Derek Long. The AIPS-98 planning competition. *AI MAgazine*, 21(2):13–32, 2000.

27 Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

28 Lawrence C Paulson. A formalisation of finite automata using hereditarily finite sets. In *International Conference on Automated Deduction*, pages 231–245. Springer, 2015.

29 John Penix, Charles Pecheur, and Klaus Havelund. Using model checking to validate ai planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard*, 1998.

30 Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting problem symmetries in state-based planners. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

31 Buser Say and Scott Sanner. Planning in factored state and action spaces with learned binarized neural network transition models. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 4815–4821, 2018.

32 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logics*, pages 424–439. Springer, 2009.

33 Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

34 Benjamin Smith, William Millar, Julia Dunphy, Yu-Wen Tung, Pandu Nayak, Ed Gamble, and Micah Clark. Validation and verification of the remote agent for spacecraft autonomy. In *1999 IEEE Aerospace Conference. Proceedings (Cat. No. 99TH8403)*, volume 1, pages 449–468. IEEE, 1999.

35 Benjamin D Smith, Martin S Feather, and Nicola Muscettola. Challenges and methods in testing the remote agent planner. In *AIPS*, pages 254–263, 2000.

36 Margaret H Smith, Gerard J Holzmann, Gordon C Cucullu, and BD Smith. Model checking autonomous planners: Even the best laid plans must be verified. In *2005 IEEE Aerospace Conference*, pages 1–11. IEEE, 2005.

37 Christoph Sprenger. A verified model checker for the modal μ-calculus in Coq. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–183. Springer, 1998.

38 Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Proceedings of the Thirty-Second AAAI Conference*

on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pages 6294–6301, 2018.

39 Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *International Joint Conference on Artificial Intelligence*, pages 1178–1185. Morgan Kaufmann Publishers, 1997.

40 Chunhan Wu, Xingyuan Zhang, and Christian Urban. A formalisation of the myhill-nerode theorem based on regular expressions (proof pearl). In *International Conference on Interactive Theorem Proving*, pages 341–356. Springer, 2011.