Automatic cross-domain task plan transfer by caching abstract skills

Khen Elimelech , **Lydia E. Kavraki** and **Moshe Y. Vardi**
Department of Computer Science, Rice University, Houston, TX 77005, USA.
{elimelech, kavraki, vardi}@rice.edu

Abstract

Solving realistic robotic task planning problems is computationally demanding. To better exploit the planning effort and reduce the future planning cost, it is important to increase the reusability of successful plans. To this end, we suggest a systematic and automatable approach for plan transfer, by rethinking the plan caching procedure. Specifically, instead of caching successful plans in their original domain, we suggest transferring them upon discovery to a dynamically-defined abstract domain and cache them as "abstract skills" there. This technique allows us to maintain a unified, standardized, and compact skill database, to avoid skill redundancy, and to support lifelong operation. Cached skills can later be reconstructed into new domains on demand, and be applied to new tasks, with no human intervention. This is made possible thanks to the novel concept of "abstraction keys." An abstraction key, when coupled with a skill, provides all the necessary information to cache it, reconstruct it, and transfer it across all domains in which it is applicable — even domains we have yet to encounter. We practically demonstrate the approach by providing two examples of such keys and explain how they can be used in a manipulation planning domain.

1 Introduction

1.1 Background

Task planning [1], often mentioned in the context of task and motion planning [2], is the problem of finding discrete plans or policies of symbolic actions, to instruct an agent to achieve a certain goal. This formulation is useful for solving various robot tasks, such as manipulation, rearrangement, and navigation, which can be specified in corresponding planning domains. The solution to such tasks is often computationally demanding, especially when considering high-dimensional robots, obstructed spaces, or complicated task specifications. Hence, to best exploit the intensive computational effort typically invested in such planning, it is important to develop techniques to maximize the reusability of successful plans. Potentially, such techniques can allow us to avoid redundant calculations while planning, and, by such, improve future planning efficiency.

Assume we found a successful solution to a certain task in a certain planning domain; if we expect to face this task again, a simple way to increase the solution reusability is to memorize or cache it for future usage. However, such caching is extremely limited, as the plan is tightly coupled with the initial state, task, and domain specifications; even a slight change in one of those aspects might render the plan invalid. Hence, to increase the computational benefits achieved from each cached solution, we must be able to generalize its applicability to other contexts, such as across similar domains and across similar tasks in the domain. Broadly, this problem is known as *skill transfer*. We note that the term "skill" can have various interpretations in the literature, conveying different information that can be inferred and transferred from past planning experience; e.g., sampling distributions for sampling-based planning [3] and motion plan segments [4; 5]. In the context of this paper, we use the term "skill" to convey a successful discrete task plan in a given domain.

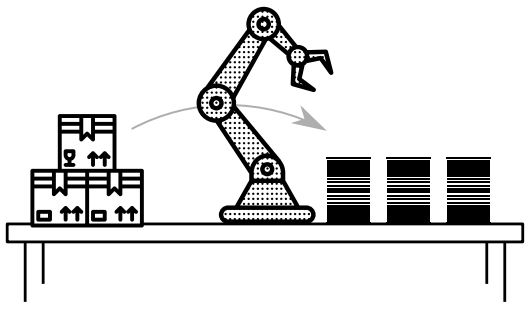

Figure 1: Skill transfer: from a box stacking plan to a can stacking plan.

Work on this paper was supported in part by NSF-IIS-1830549. This paper first appeared in the proceedings of the 2022 Workshop on the Algorithmic Foundations of Robotics (WAFR).

1.2 Motivation

Many times, and especially when considering lifelong operation of an embodied AI system, task specifications and planning domains can overlap, and show topological similarities. Further, the spatial locality that often characterizes actions in robotic domains implies that such similarities can even be found within the domain. Intuitively, we can expect these similarities to be exploitable for skill transfer. For example, we can consider a simple manipulation domain, which contains six objects — three boxes and three cans — and a robotic arm capable of picking and placing these objects in requested locations (either on the table, or on top of each other). In this domain, which is visualized in Figure 1, we care to solve two tasks: first, "stack the boxes in a certain formation" and then "stack the cans in the same formation".

It is reasonable to think that a successful plan to stack the boxes can be generalized into a successful plan to stack the cans. Needless to say, by doing so, we can dramatically reduce the computational cost of solving the second task. Yet, although this generalization might seem trivial to a human observer, it is not as straightforward for the machine. For example, to adapt the stacking plan from boxes to cans, we need to: shift the object placement locations to the other side of the table; replace each operation on a box with operation on a respective can; and replace each box pick/place action with a can pick/place action, as these can convey, e.g., different gripping tactics.

1.3 Contribution

As implied, caching skills — the process of encoding skills and saving them in memory — is essential for the ability to reuse and transfer them in the future. Accordingly, this paper introduces two interwoven contributions to increase skill reusability: (i) a novel skill caching technique, which we use as a basis to develop (ii) a novel skill transfer technique. Specifically, we suggest caching successful plans as skills in an abstract domain, from which they can be automatically generalized and transferred across new tasks and domains on demand. Both the caching and the transfer can be performed with no human intervention and with minimal additional computational effort.

Technically, the caching and transfer techniques are made possible by introducing the novel concept of "abstraction keys." According to the suggested caching approach, alongside each cached abstract skill, we shall also cache its "public abstraction key," which, when coupled with an appropriate "private key," provides the necessary information to cache the skill, reconstruct it, and transfer it to new domains (even domains we have yet to encounter). This way, skills can inherently only be transferred to domains in which they are applicable, as only those can yield a valid private key — similarly to "public key" encryption. This allows us to automatically verify the applicability of skills to tasks, and avoid futilely using them in domains in which their execution would be unsuccessful. We provide examples of two practical and generic abstraction keys, and explain how they can be used in our motivating example.

Although the two contributions are connected, they can, nonetheless, be applied separately. The suggested caching technique allows us to efficiently maintain a unified, standardized, and inherently compact database of successful plans from the past. Thus, this technique can be used even without performing transfer, to enjoy the compactness of the representation, to avoid skill redundancy, and to support lifelong operation. Appropriately, we can also consider transferring plans through the abstract domain, without caching them.

The paper is organized as follows: we begin in Section 2 with a formal problem definition. In Section 3, we provide additional motivation and an overview of our approach. Then, in Section 4, we develop and demonstrate the technique for abstract skill caching, using abstraction keys. Finally, we extend this idea, and develop the transfer technique in Section 5. While all practical details are given in the main text, we also provide summarized algorithms in Appendix A.

1.4 Related work

To the best of our knowledge, this is the first work to specifically address the scalability and compactness of skill caching in the context of robotic planning. On the other hand, the notions of skill transfer and domain abstraction have been examined extensively by the robotics and AI planning communities.

The problem of skill transfer is most often considered in the context of skill transfer learning, or the closely related concepts of imitation learning and behavior cloning [6; 7; 8]. In these problems, we try to learn a policy or plan components from demonstrations of an expert, which operates according to a hidden policy. These problems are essentially different than the one we consider here — we do not construct skills from (multiple) observations, but from other known skills, whether in our domain or another. That problem also inherently involves human intervention, something we explicitly wish to avoid. Further, the learned transfer functions in that case are coupled with the specific task and domain; this is in contrast to our abstraction-based transfer technique, which can be used to transfer each skill across multiple domains and tasks. As mentioned, utilization of past planning experience is mostly prominent in continuous planning domains, which are not the ones we consider here. Another related problem is domain adaptation [9], in which one tries to generalize knowledge from one domain to another. This adaptation, however, typically relates to machine learning problems, such as classification and regression, and not planning.

Abstraction is a widely used tool for solving task and motion planning problems, especially when considering continuous or large domains [10; 11; 12]. By imposing abstraction on a planning domain, a planning problem can be efficiently solved in a hierarchical fashion; i.e., by finding a high-level plan in the abstract domain, and, based on this plan, infer a compliant low-level plan in the robot domain. This way, we can exploit the abstract plan to induce additional constraints on the original planning domain, and, by such, restrict the search there, and improve its efficiency. Yet, our usage of abstraction is different, as we do not actively plan in the abstract domain; instead, we only use the abstract domain to represent a given plan in a compact and generalizable way. Further, hierarchical planning usually requires predefin-

ing an abstraction of the entire planning domain; in contrast, our plan abstractions are chosen dynamically based on the skill at hand, operate locally in the domain, and are invertible. Some recent works [13; 14] propose finding transferable plans by planning in a "latent space", which conveys, essentially, a learned abstraction of the planning domain. Not only is our abstraction is symbolically defined, and requires no learning, but also, as mentioned, we do not care to plan in the abstract domain. Similarly to us, other works [15; 16] also take advantage of dynamically chosen abstraction for efficient policy synthesis. These approaches, however, consider specific planning domains, and do not use the abstraction for transfer.

Finally, we should also mention that, in the context of reinforcement learning and Markov decision process, abstraction selection is an integral part in improving the accuracy of skill learning (see, e.g., [17]); this is, of course, a vastly different context than the classical planning scenario we consider here.

2 Problem definition

Let us begin by formally defining *task planning domains*.

Definition 1. A *task planning domain* $D \doteq (\mathcal{S}, \mathcal{A}, \mathcal{T})$ is comprised of a state space $\mathcal{S}$ (continuous or discrete), an action space $\mathcal{A}$, and a set $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ of discrete transitions. Transitions in the domain are *deterministic*, meaning, $\forall (\boldsymbol{S}, a, \boldsymbol{S}') \in \mathcal{T}$, $\nexists \boldsymbol{S}'' \in \mathcal{S}$, such that $(\boldsymbol{S}, a, \boldsymbol{S}'') \in \mathcal{T}$. An *execution* $\boldsymbol{E}$ is a sequence of alternating states and actions, i.e.,

$$\boldsymbol{E} \doteq (\boldsymbol{S}_0, a_1, \boldsymbol{S}_1, a_2, \dots, a_n, \boldsymbol{S}_n)\,. \tag{1}$$

The execution $\boldsymbol{E}$ is *feasible* in the domain if $\forall i \in \{1, \dots, n\}$, $(\boldsymbol{S}_{i-1}, a_i, \boldsymbol{S}_i) \in \mathcal{T}$. In that case, we say that $\boldsymbol{E}$ is induced by the action sequence $(a_1, \dots, a_n)$.

From now on, we will refer to a task planning domain simply as a "domain". A task can be thought of as a collection of constraints on states and/or actions of a future execution. Such constraints can specify, e.g., to reach or globally avoid certain regions, or convey temporal, or cost-related requirements. For a given task, we care to find a *plan* (i.e., a sequence of actions), to be applied from the current state, whose execution is feasible in the domain and satisfies the task constraints. For the simplicity of presentation, we focus here on "classical tasks", which only require us to reach a specified goal state. We formally define the "classical task planning" problem as follows:

Definition 2. A classical task planning problem $P \doteq (D, \boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}})$ is comprised of a domain D, a start state $\boldsymbol{S}_{\text{start}} \in \mathcal{S}$, and a goal state $\boldsymbol{S}_{\text{goal}} \in \mathcal{S}$. To solve P, we seek a plan, i.e., a sequence of actions $(a_1, \dots, a_n) \in \mathcal{A}^n$, that, when applied from $\boldsymbol{S}_{\text{start}}$, induces a feasible execution in D that terminates at $\boldsymbol{S}_{\text{goal}}$. A successful plan that solves P is referred to here as a *skill*.

To be able to reuse a skill, e.g., if the task is expected to repeat in the future, it surely must be cached in memory. Hence, assume we previously found and cached a skill that solves a planning problem P, and now wish to solve another problem P'. The planning domains of the two problems may be equal, but, to not impose any restrictions, we shall assume each task is defined in its own domain. To avoid solving P' from scratch, and potentially reduce the computational cost of its solution, we can attempt to perform *skill transfer* between the problems.

Definition 3. Consider a skill κ, which solves a planning problem $P \doteq (D, \boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}})$. Also consider another problem $P' \doteq (D', \boldsymbol{S}'_{\text{start}}, \boldsymbol{S}'_{\text{goal}})$, for which we seek a solution. A *transfer function* between domains D and D' is a function of the form

$$\texttt{transfer-D-to-D'} : \{\text{plans in } D\} \rightarrow \{\text{plans in } D'\}. \tag{2}$$

We say that *skill transfer* between the problems can be performed successfully, if we can find such a transfer function, such that

$$\texttt{transfer-D-to-D'}\,(\kappa) = \kappa' \tag{3}$$

is a solution of P'.

The objective of this work is to develop a systematic, automatable, and scalable technique to perform such transfer. This technique should also be generic, i.e., not coupled with any specific task or domain.

3 Scalable skill transfer through an abstract domain

We now wish to examine the scalability of the skill transfer problem; thus, assume more generally that we care to transfer multiple skills across multiple planning problems in multiple domains.

As mentioned, a necessary step to allow future skill transfer is caching skills upon discovery. When considering planning problems across multiple domains, we shall essentially maintain a separate skill cache for each domain, according to the planning problems we previously solved in it. Hence, to take advantage of transfer when solving a new task, we would have to access the skill cache of every domain, query it for potentially relevant skills, and attempt to transfer them, by finding and applying appropriate transfer functions. If a transfer of a skill is successful, we shall cache a copy of it as a new skill in the destination domain. Unfortunately, this naive approach not only scales poorly, and leads to computational and memory redundancy, but also leads to an undesirable explicit coupling between the domains. This scenario is demonstrated in Fig. 2a.

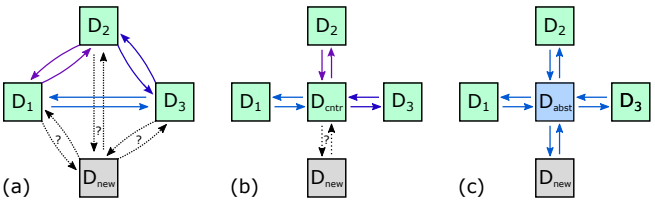

Figure 2: Skill (plan) transfer across domains: (a) without a central domain; (b) with a central domain; (c) with an abstract central domain. In green — known domains; in gray — unseen domains; arrows represent skill transfer functions between the domains.

Addressing the previous concerns, we wish to offer a preferable approach for skill transfer, by altering the basic skill caching procedure. Specifically, instead of caching skills as plans in their original domain, we suggest to transfer them upon discovery to a new and "central" domain, and cache them as skills there; this way, we can maintain a unified and standardized skill cache, and avoid the explicit coupling between domains. This idea is demonstrated in Figure 2b.

With this approach, skill transfer is performed in two independent stages: (i) on each successful task planning, the discovered plan is mapped to a plan in (i.e., transferred to) a central domain, and cached there as a skill, for future use; (ii) on demand, we may access the skills previously cached in the central cache, examine and transfer them to new domains, to solve new planning problems. Accordingly, the skill transfer process can be presented using two functions

$$\texttt{transfer-D-to-D}_{\text{center}}\texttt{:}\quad \{\text{plans in } D\} \to \{\text{plans in } D_{\text{center}}\}, \tag{4}$$

$$\texttt{transfer-D}_{\text{center}}\texttt{-to-D}'\texttt{:}\quad \{\text{plans in } D_{\text{center}}\} \to \{\text{plans in } D'\}, \tag{5}$$

such that

$$\texttt{transfer-D-to-D}_{\text{center}}(\kappa) = \kappa_{\text{center}}, \tag{6}$$

$$\texttt{transfer-D}_{\text{center}}\texttt{-to-D}'(\kappa_{\text{center}}) = \kappa'. \tag{7}$$

Despite the clear advantages of decoupling the domains and avoiding skill redundancy, reliance on this approach also introduces new challenges, as (i) finding a central caching domain is not trivial, especially when considering infinite domains; (ii) transfer through an arbitrary domain might compromise the generalizability of skills, or jeopardize the potential feasibility of transferred skills in new domains; (iii) we would still need to infer transfer functions between every domain to the central domain.

As we shall demonstrate ahead, all these challenges can be mitigated with a "smart" definition of the central caching domain. Specifically, there is no need to predefine the central domain explicitly; instead, we can let each skill that we wish to transfer to dynamically determine the domain it should be cached in (answering challenge (i)). Further, by choosing an *abstract* central domain, skills can be cached *compactly*, while still allowing lossless reconstruction into their original domain, and intuitive transfer into new domains (answering challenge (ii)). This choice will also allow us to rely on a unified pair of transfer functions (4)-(5) for each skill, instead of having to infer them for each new domain separately (answering challenge (iii)). These benefits are demonstrated in Figure 2c.

In the following sections we formulate these ideas, and provide a structured and automatable technique to define such a central abstract domain, and use it for skill transfer.

4 Skill abstraction and caching

Let us now develop the first step in our transfer approach: transferring skills to an abstract domain. Thus, assume we acquired a skill (i.e., found a successful plan) $\kappa \doteq (a_1, \ldots, a_n)$, which solves the planning problem $P \doteq (D, \boldsymbol{S}_0, \boldsymbol{S}_n)$, and let $\boldsymbol{E}$ mark the execution induced by this skill in domain D, as defined in (1).

4.1 Representing transferable skills as traces of states

To transfer this skill to a new domain, a typical transfer approach might suggest to translate the sequence of actions to the new context. Yet, we recognize that while mapping between state spaces is often intuitive, mapping between action spaces, without compromising the state connectivity, is non-trivial. In relation to our specific technique, defining a (reconstructable) action abstraction is even less trivial. This implies that, with a typical transfer approach, we would only be able to transfer skills across domains which share the same (or a very similar) action space. We surely do not want to limited by such a constraint. Also, even if such action transfer can be performed successfully, this approach would make it challenging to automatically determine skill applicability to new tasks, without performing explicit action evaluation.

Thus, unlike standard caching and transfer approaches, we suggest here to represent the skill using the *trace of states* from its execution, and focus on transferring *it* into new domains. We mark this trace as $\mathfrak{S}(\boldsymbol{E}) \doteq (\boldsymbol{S}_0, \ldots, \boldsymbol{S}_n)$. As we learn ahead, this choice of state-based skill representation is essential to allow automated and wide skill generalization and transfer to unseen contexts. By doing so, for a transfer to be successful, we only care for the transferred states to maintain their connectivity from the original domain, and do not need to care for the semantics of actions in other domains. Each domain we transfer a skill to may independently recover the sequence of actions that connect the transferred state trace in it. Unless specified otherwise, from now on, when referring to the "skill's trace", we refer to $\mathfrak{S}(\boldsymbol{E})$.

4.2 Abstraction keys

As suggested before, we wish to transfer the skill's trace to an abstract domain. As a baseline, we demand from our abstraction mechanism to allow perfect reconstruction of the skill's original trace. We also want the abstract skill's representation to be at least as compact as the original trace, to maintain memory efficiency. These objectives can be achieved by finding a pair of inverse functions

$$\texttt{project}\colon \mathcal{S} \to \Xi, \tag{8}$$

$$\texttt{reconst}\colon \Xi \to \mathcal{S}, \tag{9}$$

such that

$$\texttt{dim}(\Xi) \leq \texttt{dim}(\mathcal{S}), \tag{10}$$

$$\texttt{reconst}(\texttt{project}(\boldsymbol{S})) = \boldsymbol{S}, \quad \forall \boldsymbol{S} \in \mathfrak{S}(\boldsymbol{E}). \tag{11}$$

These functions represent state "projection" and "reconstruction", respectively. The first condition (in (10)) verifies that the selected projection function maps each state to a more compact representation. The second condition (in (11)) verifies that the reconstruction from each projected state is *lossless*.

Such functions allow us to efficiently represent a skill's trace $(\boldsymbol{S}_0, \ldots, \boldsymbol{S}_n)$ using the compact state projections:

$$\texttt{project}(\boldsymbol{S}_0, \ldots, \boldsymbol{S}_n) \doteq (\texttt{project}(\boldsymbol{S}_0), \ldots, \texttt{project}(\boldsymbol{S}_n)) = (\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n). \tag{12}$$

Later, the original trace can be reconstructed on demand by applying the reconstruction function:

$$\texttt{reconst}\left(\boldsymbol{\xi}_0,\ldots,\boldsymbol{\xi}_n\right) \doteq \left(\texttt{reconst}\left(\boldsymbol{\xi}_1\right),\ldots,\texttt{reconst}\left(\boldsymbol{\xi}_n\right)\right) = \left(\boldsymbol{S}_0,\ldots,\boldsymbol{S}_n\right). \quad (13)$$

Since Ξ is a projection of the original domain's state space, it can be considered an *abstract state space*. The projected trace can hence inherently represent a skill in an abstract domain $D_{\text{abstract}} \doteq (\Xi, \mathcal{A}_{\text{abstract}}, \mathcal{T}_{\text{abstract}})$, where the action and transition sets simply reflect abstract actions connecting the consecutive states in the trace, i.e., $\mathcal{A}_{\text{abstract}} \doteq \{\alpha_i\}_{i=1}^n$, and $\mathcal{T}_{\text{abstract}} \doteq \{(\boldsymbol{\xi}_{i-1}, \alpha_i, \boldsymbol{\xi}_i)\}_{i=1}^n$. We hence refer to such pair of functions as an *abstraction key*. A skill can potentially be abstracted using different such keys, and the same key can be applied to different skills. The choice of key implicitly determines the abstract domain the skill's trace is transferred to, while the "skill transfer function" (defined in (4)) is implied by the key's projection function.

4.3 Parametric abstraction keys

Searching directly for compatible abstraction keys in order to perform the skill transfer is not computationally practical. Hence, instead, we suggest to examine a collection of predefined parametric abstraction keys

$$\texttt{project}_p\colon \mathcal{S} \to \Xi, \quad (14)$$

$$\texttt{reconst}_p\colon \Xi \to \mathcal{S}, \quad (15)$$

and look for an appropriate parameter value to transfer our skill of interest.

Intuitively, we can consider every parametric key to operate on a property of the input state (e.g., "location" or "type of object"), and the parameter p to reference a specific value of this property (e.g., "on the table" or "box"). This reference can be "removed" from the state with the projection function, and "incorporated back" into it with the reconstruction function. Naturally, the definition of the parametric key should also specify the valid values for p ("the parameter space") that these functions can handle. The parameter space is generally defined in relation to the input state; we mark the parameter space, for an input state $\boldsymbol{S} \in \mathcal{S}$, as $\mathcal{P}_{\boldsymbol{S}}$.

To project and cache a skill's trace with a certain parametric key, we need to find a *single* parameter value p, which can be used to project *all* states in the trace; together, the parametric functions and the valid choice of parameter, convey an abstraction key for the skill. More specifically, since the parametric functions are known to all and predefined, we can say that they convey a *public abstraction key* for traces from the state space.

Definition 4. Considering a state space $\mathcal{S}$, a *public abstraction key PubKey* $\doteq \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_{\boldsymbol{S}}\right)$ is comprised of a state projection function $\texttt{project}_p$, as defined in (14); a state reconstruction function $\texttt{reconst}_p$, as defined in (15); and the state-dependent parameter space $\mathcal{P}_{\boldsymbol{S}}$, such that $p \in \mathcal{P}_{\boldsymbol{S}}$.

The parameter value, which is derived from and for the specific skill, represents the skill's *private abstraction key*.

Definition 5. Consider a skill with state trace $\mathfrak{S}(\boldsymbol{E})$, and a public abstraction key *PubKey* $\doteq \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_{\boldsymbol{S}}\right)$. A parameter choice p is a valid *private abstraction key* for this skill, if

$$\forall \boldsymbol{S} \in \mathfrak{S}(\boldsymbol{E}), \quad p \in \mathcal{P}_{\boldsymbol{S}} \quad \text{and} \quad \texttt{reconst}_p\left(\texttt{project}_p\left(\boldsymbol{S}\right)\right) = \boldsymbol{S}. \quad (16)$$

4.4 Abstract skill caching

If a valid combination of a public and private key exists, we can use it to transfer (i.e., project the trace of) the skill into the key-imposed abstract domain D_{abstract}, where it can be cached. We refer to the skill represented by this abstract state trace $\texttt{project}_p\left(\mathfrak{S}(\boldsymbol{E})\right)$ in that abstract domain, alongside its abstraction key, as an *abstract skill*. In fact, as we later show in Section 5, to allow future transfer of the abstract skill into new domains, or reconstruction of the original skill, it is sufficient to cache *only* its public key with the abstract state trace. Nonetheless, we can always cache also the original action trace, and/or the private key, to be used as reference. The caching approach is later summarized in Algorithm 1 in Appendix A.

We note that, for a certain skill, there could be multiple valid key combinations. In that case, the choice of keys might affect the compactness of the cached abstract skill. Surely, we would always prefer the key that achieves the most compact representation. As a rule of thumb, more information contained in the private key reflects a more compact abstract trace (and, by such, a higher generalization potential); this will be demonstrated in the exemplary abstraction keys, to be given next.

4.5 Exemplary abstraction key: attention

This parametric abstraction key allows us to focus our attention only on a subset of m state variables, while exporting the remainder to the parameter p. The parameter space of a state vector $\boldsymbol{S}$ is hence defined as

$$\mathcal{P}_{\boldsymbol{S}} \doteq \left\{(m \text{ variable indexes, value of other variables})\right\} \equiv \left\{ \left(\mathcal{I}, \boldsymbol{S}[\neg\mathcal{I}]\right) \mid \mathcal{I} \subseteq \{1,\ldots,|\boldsymbol{S}|\},\ |\mathcal{I}| = m,\ \neg\mathcal{I} \doteq \{1,\ldots,|\boldsymbol{S}|\} \setminus \mathcal{I} \right\}, \quad (17)$$

where $x[y]$ marks the y-th element of x. Each parameter $p \in \mathcal{P}_{\boldsymbol{S}}$ is comprised of two components: a set containing the indexes of the m state variables on which we want to put attention (to be used by the projection function); and the value of the remainder $|\boldsymbol{S}| - m$ state variables (to allow reconstruction).

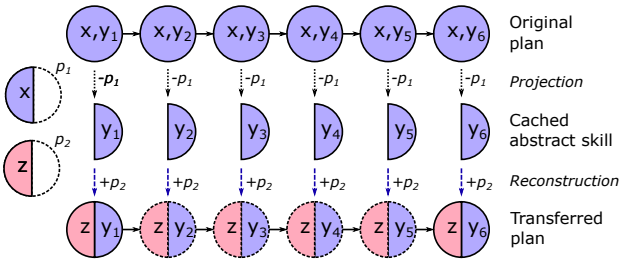

Figure 3: Transferring plans across domains using "attention".

Accordingly, the projection and reconstruction functions are defined as:

$$\texttt{m-atten-project}_p\left(\boldsymbol{S}\right) = \boldsymbol{S}[\mathcal{I}], \tag{18}$$

$$\texttt{m-atten-reconst}_p\left(\boldsymbol{\xi}\right) = \boldsymbol{S}, \text{ such that} \tag{19}$$
$$\boldsymbol{S}[\mathcal{I}] \leftarrow \boldsymbol{\xi} \wedge \boldsymbol{S}[\neg\mathcal{I}] \leftarrow values,$$

where $\mathcal{I} \doteq p[1]$, $values \doteq p[2]$.

We recall that to project a skill's trace $\mathfrak{S}\left(\boldsymbol{E}\right)$ using a parametric key, we should use the same parameter value for all states. Thus, a parameter choice $p = (\mathcal{I}, \boldsymbol{S}[\neg\mathcal{I}])$ would be valid as a private key if and only if $\boldsymbol{S}[\neg\mathcal{I}]$ (the value of the state variables with indexes $\neg\mathcal{I}$) remains constant $\forall \boldsymbol{S} \in \mathfrak{S}\left(\boldsymbol{E}\right)$. This means that, practically, this parametric key is used to compactly cache a skill's state trace as a trace of "smaller" states, containing only the state variables which are affected by the skill; the "constant variables", which repeat in every state without changing their values, can be exported to the skill's private key.

This key is hence useful for abstraction and transfer of "local skills", which are comprised of actions that present spatial locality, and only affect a small subset of the state variables. This is often the case in robotic domains, as evident, in the manipulation domain provided in the introduction; there, repositioning of an object is a local skill, as it does not affect the positions of the other objects. A visualization of this key is provided in Figure 3.

4.6 Exemplary abstraction key: symbol stripping

This parametric abstraction key allows us to focus on function over form, by extracting the repeating symbols from states in the state trace. As far as this paper is concerned, a symbol may be a certain (often categorical) value val, assigned to a state variable s_i, or a sequence of m such values $(val_1, \ldots, val_m)$, assigned respectively to a sequence of variables $(s_i, \ldots, s_{i+m-1})$ in the state vector. The parameter space of $\boldsymbol{S}$ is hence defined as

$$\begin{aligned}&\mathcal{P}_{\boldsymbol{S}} \doteq \{\text{symbols of length } m \text{ that appear in } \boldsymbol{S}\}\\ &\equiv \{(\boldsymbol{S}[i], \ldots, \boldsymbol{S}[i+m-1]) \mid i \in \{1, \ldots, |\boldsymbol{S}|-m+1\}\}.\end{aligned} \tag{20}$$

On projection, we can "strip" a state from a chosen symbol $p \in \mathcal{P}_{\boldsymbol{S}}$, by replacing all of its appearances with "placeholders". Later, on reconstruction, we shall simply repopulate those placeholders with p.

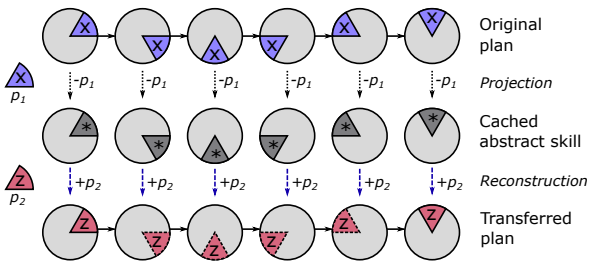

Figure 4: Transferring plans across domains using "symbol stripping".

Accordingly, the projection and reconstruction functions are defined as:

$$\texttt{m-symb-project}_p\left(\boldsymbol{S}\right) = \text{replace each instance of } p \text{ in } \boldsymbol{S} \text{ with } *, \tag{21}$$

$$\texttt{m-symb-reconst}_p\left(\boldsymbol{\xi}\right) = \text{replace each instance of } * \text{ in } \boldsymbol{\xi} \text{ with } p. \tag{22}$$

We again recall that to project a skill's trace $\mathfrak{S}\left(\boldsymbol{E}\right)$, we require the same parameter value for all states. Thus, a parameter choice p would be valid as a private key if and only if p is a symbol that appears in all states in $\mathfrak{S}\left(\boldsymbol{E}\right)$. This key is hence also relevant to the manipulation domain provided in the introduction; there, it can be used to cache an object stacking plan, with no explicit reference to the specific object type. A visualization of this key is provided in Figure 4.

We note that while both "symbol striping" and "attention" are useful abstraction keys when there are values that repeat in each state in the trace, each of them is used for different scenarios. "Symbol stripping" requires the values to appear in (the same) sequence, though this sequence may appear in a different index in each state; "attention" does not require the values to appear in sequence, though their index in each state must be fixed.

5 Abstract skill reconstruction and transfer

After establishing the skill abstraction process, we are ready to explain the second step in our approach: transferring a cached abstract skill to a new domain.

Thus, consider an abstract skill represented by the state trace $(\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n)$, and cached using a public key $\left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_s\right)$, and a private key $\hat{p}$. We are now interested in transferring this abstract skill into a new domain D, in order to solve a new ("classical") task planning problem $P \doteq (D, \boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}})$. Our parametric abstraction approach provides us with a structured and straightforward method to perform such transfer — by looking for a new private key p that allows appropriate reconstruction of the abstract skill for P.

5.1 Abstract skill applicability

Of course, not always an appropriate private key p for such reconstruction exists. If we can reconstruct the abstract skill's trace into a trace in D that matches P, we say that the abstract skill is *applicable* to the problem.

Definition 6. Consider an abstract skill represented by a state trace $(\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n)$, which was cached with a public key $\left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_s\right)$. The skill is *applicable* to the "classical" task planning problem $P \doteq (D, \boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}})$, if exists a private key $p \in \mathcal{P}_{\boldsymbol{S}_{\text{start}}} \cap \mathcal{P}_{\boldsymbol{S}_{\text{goal}}}$ such that

$$\texttt{project}_p\left(\boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}}\right) = (\boldsymbol{\xi}_0, \boldsymbol{\xi}_n) \text{ or } \texttt{reconst}_p\left(\boldsymbol{\xi}_0, \boldsymbol{\xi}_n\right) = (\boldsymbol{S}_{\text{start}}, \boldsymbol{S}_{\text{goal}}). \tag{23}$$

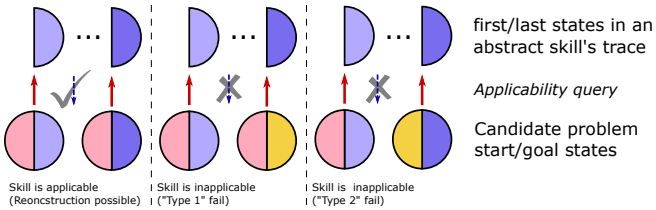

Figure 5: Skill applicability test.

Meaning, to determine applicability with this type of tasks, we only need to examine the mapping between the first and last states of the skill's trace, and the start and goal states of the planning problem. Accordingly, the skill is inapplicable to the task if: (i) $\boldsymbol{S}_{\text{start}}$ cannot be projected to $\boldsymbol{\xi}_0$, using any choice of parameter, and/or $\boldsymbol{S}_{\text{goal}}$ cannot be projected to $\boldsymbol{\xi}_n$, using any choice of parameter; or (ii) $\boldsymbol{S}_{\text{start}}$ can be projected to $\boldsymbol{\xi}_0$, using a certain parameter, and $\boldsymbol{S}_{\text{goal}}$ can be projected to $\boldsymbol{\xi}_n$, using a certain parameter, but not using the same parameter. This is visualized in Figure 5. Note that for more complex tasks, determining applicability may require examination of the entire trace.

Determining skill applicability

As explained, to determine the applicability of a cached abstract skill to the problem, we shall reason about the existence of an appropriate private key p for skill reconstruction. Conveniently, if such p exists, it can be automatically derived from the problem specification; in particular, we can use $\boldsymbol{S}_{\text{start}}$ and $\boldsymbol{S}_{\text{goal}}$ as *anchor states*, constraining the parametric mapping.

As we recall, the definition of a public key must specify the parameter space $\mathcal{P}_{\boldsymbol{S}}$ (of a state $\boldsymbol{S}$) for a valid projection. Intuitively, forcing the projection of $\boldsymbol{S}$ to return a specific value $\boldsymbol{\xi}$ imposes additional constraints on the parameter space; for a certain public key, we can mark the set of valid parameters that transfer between a state $\boldsymbol{S}$ and an abstract state $\boldsymbol{\xi}$ as $\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}} (\subseteq \mathcal{P}_{\boldsymbol{S}})$. For many such keys, this subset $\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}}$ can be symbolically defined and encoded in the public key as a complementary component, with no additional effort. This claim is evident in the two exemplary keys provided before, in which $\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}}$ can be easily inferred by imposing an additional constraint on $\mathcal{P}_{\boldsymbol{S}}$. I.e., for "attention":

$$\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}} \doteq \left\{ (\mathcal{I}, \boldsymbol{S}_{\neg\mathcal{I}}) \in \mathcal{P}_{\boldsymbol{S}} \mid \boldsymbol{S}_{\mathcal{I}} = \boldsymbol{\xi} \right\}; \tag{24}$$

for "symbol stripping":

$$\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}} \doteq \{ p \in \mathcal{P}_{\boldsymbol{S}} \mid p \text{ does not appear in } \boldsymbol{\xi} \}. \tag{25}$$

If the definition of $\mathcal{P}_{\boldsymbol{S}}^{\boldsymbol{\xi}}$ is available for our public key, the applicability test can be performed instantly, by examining if the intersection $\mathcal{P}_{\boldsymbol{S}_{\text{start}}}^{\boldsymbol{\xi}_0} \cap \mathcal{P}_{\boldsymbol{S}_{\text{goal}}}^{\boldsymbol{\xi}_n}$ is not empty. Otherwise, we can test for applicability by actively searching for an appropriate parameter $p \in \mathcal{P}_{\boldsymbol{S}_{\text{start}}} \cap \mathcal{P}_{\boldsymbol{S}_{\text{goal}}}$, which satisfies (23). These steps are summarized in Algorithm 2 in Appendix A.

5.2 Skill feasibility

If an abstract skill is applicable, we can transfer its trace to the task domain, by reconstructing it using the private key p we derived from the anchors. To complete the transfer, we need to infer a feasible plan of actions $(a_1, \ldots, a_n)$ in the task domain, which connect the states in the reconstructed state trace. If all inter-state transitions are feasible, we say that the abstract skill is *feasible*.

Definition 7. Consider a classical planning problem in domain $D \doteq (\mathcal{S}, \mathcal{A}, \mathcal{T})$, and a state trace $(\boldsymbol{S}_0, \ldots, \boldsymbol{S}_n)$ in $\mathcal{S}$, reconstructed from an abstract skill's trace. This abstract skill is *feasible*, if exists a sequence $(a_1, \ldots, a_n) \subseteq \mathcal{A}^n$ of actions, such that $(\boldsymbol{S}_{i-1}, a_i, \boldsymbol{S}_i) \in \mathcal{T}$, $\forall i \in \{1, \ldots, n\}$.

Again, for the simplicity of this paper, we assume every inter-state transition is realized with a single action in D. Then, inferring the actions can be easily done, by looking for an action whose preconditions are satisfied by $\boldsymbol{S}_i$ and the effect is expressed in $\boldsymbol{S}_{i+1}$, $\forall i$. More generally, a transition can be realized with a sequence of multiple such actions ("sub-plan"), as to be explored in future work.

5.3 Skill transfer through an abstract domain: recap

We can summarize the suggested skill transfer technique with the following steps: (i) transfer the skill's state trace from its original domain to an abstract domain by selecting valid public and private keys and cache it; (ii-a) on demand, transfer the cached skill's trace into any new domain, using the original public key and an alternative private key, derived from the new task specification; (ii-b) infer the actions in that domain to transition across the states in that trace. The steps for transferring the cached skill are summarized in Algorithm 3 in Appendix A.

We can also now confirm the approach advantages, initially suggested in Section 3 and Figure 3. Since we dynamically choose the abstraction keys according to the skill itself, every skill is, in fact, transferred through its own central abstract domain D_{abstract}; this domain, and the transfer function (4) that leads to it, are implicitly determined by the projection function in the abstraction key.

Further, thanks to this parametric abstraction technique, every cached skill, through its public key, inherently defines the single transfer function (5) that allows to transfer it from the abstract domain to any domain in which it is applicable (with a proper choice of private key). Hence, the approach allows us to enjoy skill transfer even across domains we have yet to encounter — with no generalization error, and with no additional effort. This technique inherently verifies that skills can only be reconstructed from the cache by problems for which they are applicable, in a similar way to a "public key" encryption. I.e., if we try to provide the skill's public key with inappropriate anchors, they will be rejected, and we will not be able to to reconstruct the skill. This ensure we do not futilely consider using them where their execution would be unsuccessful.

When is this approach useful?

Nevertheless, with our suggested transfer technique, we cannot transfer skills across completely arbitrary domains, as, in that case, the anchors might not convey sufficient information for reconstruction. This technique only allows to transfer skills across domains and tasks that demonstrate symmetry via some "property transform", which we can account for with a proper selection of a public key. Fortunately, this type of symmetry covers many practical and intuitive symmetries, as demonstrated from our introductory manipulation example, in which we cared to transfer a box stacking plan to a can stacking plan. In that scenario, for example, we can practically use the "attention key" to generalize the existing plan to the new objects of interest, and the "symbol stripping" key, to generalize the plan to apply to new object types (i.e., cans and not boxes).

6 Conclusion

In this paper we introduced a general and automatable technique for transferring skills (i.e., successful plans) across similar tasks and domains, by transferring them through a shared abstract domain. Such a domain can be dynamically defined by the skill itself, and without human intervention, using parametric abstraction keys. By separating the transfer into two stages, we can compactly cache skills for future usage (as state traces) in an abstract domain, instead of caching them in their original domain. This approach allows us to maintain a unified skill cache, decouple the domains, increase transfer scalibility, and avoid computational redundancy. We demonstrated the approach in the context of robotic manipulation planning, and provided two practical abstraction keys that can be used for skill transfer in that domain: "attention" and "symbol stripping". The underlying concept is surely applicable in even more complicated scenarios, which we could not discuss due to the limited scope of this paper.

This paper intends to lay the foundations for various future extensions. For example, since we examine a state-by-state abstraction, rather than a "full plan" abstraction, our framework naturally allows to break skills into components; such components can then be automatically recomposed in a hierarchical fashion into new skills. The state-based transfer technique can also potentially allow us to apply the transfer to continuous motion plans, or policies. Finally, when the state is "complicated", e.g., an image, we can potentially *learn* the abstraction keys (i.e., state projections), say, using neural networks. Arguably, learning such state projections would be easier and more versatile than direct transfer learning.

References

[1] Erez Karpas and Daniele Magazzeni. Automated planning for robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1):417–439, 2020.

[2] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):265–293, 2021.

[3] Constantinos Chamzas, Zachary Kingston, Carlos Quintero-Peña, Anshumali Shrivastava, and Lydia E. Kavraki. Learning Sampling Distributions Using Local 3D Workspace Decompositions for Motion Planning in High Dimensions. In *IEEE International Conference on Robotics and Automation*, pages 1283–1289, June 2021.

[4] Èric Pairet, Constantinos Chamzas, Yvan R. Petillot, and Lydia E. Kavraki. Path planning for manipulation using experience-driven random trees. *IEEE Robotics and Automation Letters*, 6(2):3295–3302, April 2021.

[5] Zachary Kingston, Constantinos Chamzas, and Lydia E. Kavraki. Using experience to improve constrained planning on foliations for multi-modal problems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 6922–6927, September 2021.

[6] Yueyue Liu, Zhijun Li, Huaping Liu, and Zhen Kan. Skill transfer learning for autonomous robots and human–robot cooperation: A survey. *Robotics and Autonomous Systems*, 128:103515, 2020.

[7] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 4950–4957. AAAI Press, 2018.

[8] Anahita Mohseni-Kabir, Changshuo Li, Victoria Wu, Daniel Miller, Benjamin Hylak, Sonia Chernova, Dmitry Berenson, Candace Sidner, and Charles Rich. Simultaneous learning of hierarchy and primitives for complex robot tasks. *Auton. Robots*, 43(4):859–874, April 2019.

[9] Sinno Jialin Pan, Ivor W. Tsang, James T. Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2011.

[10] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011.

[11] William Vega-Brown and Nicholas Roy. Admissible abstractions for near-optimal task and motion planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, page 4852–4859. AAAI Press, 2018.

[12] Danny Driess, Ozgur Oguz, and Marc Toussaint. Hierarchical task and motion planning using logic-geometric programming (HLGP). RSS Workshop on Robust Task and Motion Planning, 2019.

[13] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. *Proceedings of the AAAI Conference on Artificial Intelligence*, April 2018.

[14] Kevin Xie, Homanga Bharadhwaj, Danijar Hafner, Animesh Garg, and Florian Shkurti. Latent skill planning for exploration and transfer. In *International Conference on Learning Representations*, 2021.

[15] Keliang He, Morteza Lahijanian, Lydia E. Kavraki, and Moshe Y. Vardi. Automated abstraction of manipulation domains for cost-based reactive synthesis. *IEEE Robotics and Automation Letters*, 4(2):285–292, 2019.

[16] Jonathan A. DeCastro, Vasumathi Raman, and Hadas Kress-Gazit. Dynamics-driven adaptive abstraction for reactive high-level mission and motion planning. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 369–376, 2015.

[17] George Konidaris and Andrew Barto. Efficient skill learning using abstraction selection. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, page 1107–1112, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

A Algorithms

Algorithm 1: Caching a skill in an abstract domain.

Algorithm `cachSkill`
  Inputs:
    A state trace from the skill execution $\mathfrak{S}(\boldsymbol{E})$
    A public abstraction key
     $\mathit{PubKey} = \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_s\right)$
  Output:
    Caching success flag

  $validPrivateKeys \leftarrow \bigcap_{\boldsymbol{S} \in \mathfrak{S}(\boldsymbol{E})} \mathcal{P}_{\boldsymbol{S}}$
  if $validPrivateKeys \neq \emptyset$ **then**
    Select a key p from $validPrivateKeys$
    $abstractTrace \leftarrow \texttt{project}_p\left(\mathfrak{S}(\boldsymbol{E})\right)$
    $DataBase$.`add`($abstractTrace$, *PubKey*)
    return True `// caching succeeded`
  end
  return False `// caching failed, try another public key`

Algorithm 2: Get a private key for reconstruction (applicability test).

Algorithm `getPrivateKey`
  Inputs:
    An abstract skill's state trace $(\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n)$
    A start state $\boldsymbol{S}_{\text{start}}$
    A task goal state $\boldsymbol{S}_{\text{goal}}$
    A public key
     $\mathit{PubKey} = \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_s\right)$
  Output:
    If the abstract skill is applicable, returns a private key for reconstruction; otherwise, returns **False**

  if $\left|\mathcal{P}^{\boldsymbol{\xi}_0}_{\boldsymbol{S}_{start}}\right| = 1$ **then**
    $p \leftarrow \mathcal{P}^{\boldsymbol{\xi}_0}_{\boldsymbol{S}_{\text{start}}}$
    if $\texttt{project}_p\left(\boldsymbol{S}_{goal}\right) = \boldsymbol{\xi}_n$ *or* $\texttt{reconst}_p\left(\boldsymbol{\xi}_n\right) = \boldsymbol{S}_{goal}$ **then**
      return p `// applicable`
    end
  end
  else if $\left|\mathcal{P}^{\boldsymbol{\xi}_0}_{\boldsymbol{S}_{start}}\right| > 1$ **then**
    $intersection \leftarrow \mathcal{P}^{\boldsymbol{\xi}_n}_{\boldsymbol{S}_{\text{goal}}} \cap \mathcal{P}^{\boldsymbol{\xi}_0}_{\boldsymbol{S}_{\text{start}}}$
    if $intersection \neq \emptyset$ **then**
      Select a key p from $intersection$
      return p `// applicable`
    end
  end
  return False `// skill inapplicable`

Algorithm 3: Reconstruction of a cached abstract skill.

Algorithm `reconstructSkill`
  Inputs:
    An abstract skill's state trace $(\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n)$
    A public key
     $\mathit{PubKey} = \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}_s\right)$
    A "classical" task planning problem
     $P \doteq (D,\ \boldsymbol{S}_{\text{start}},\ \boldsymbol{S}_{\text{goal}})$
  Output:
    A plan that solves the given problem, if reconstruction succeeded; otherwise, returns **False**

  $p \leftarrow$ `getPrivateKey`$(\boldsymbol{\xi}_0, \boldsymbol{\xi}_n, \boldsymbol{S}_{start}, \boldsymbol{S}_{goal}, \mathit{PubKey})$
  if $p \neq$ **False then** `// skill is applicable`
    $states \leftarrow \texttt{reconst}_p\left(\boldsymbol{\xi}_0, \ldots, \boldsymbol{\xi}_n\right)$ `// reconstruct state trace`
    $actions \leftarrow$ `recoverActions`$(states, D)$ `// recover actions`
    if $actions \neq$ **False then**
      return $actions$
    end
  end
  return False

Procedure `recoverActions`
  Inputs:
    A state trace $(\boldsymbol{S}_0, \ldots, \boldsymbol{S}_n)$
    A planning domain D
  Output:
    If feasible, returns a sequence of actions that follows the states in the trace; otherwise, returns **False**

  for $i \in \{1, \ldots, n\}$ **do**
    if $\exists$ *an action* $a \in \mathcal{A}$ *between* $\boldsymbol{S}_{i-1}$ *and* $\boldsymbol{S}_i$ **then**
      $a_i \leftarrow a$
    end
    else
      return False `// unfeasible`
    end
  end
  return $(a_1, \ldots, a_n)$ `// feasible`