Physically Feasible Repair of Reactive, Linear Temporal Logic-Based, High-Level Tasks

Adam Pacheco and Hadas Kress-Gazit, Fellow, IEEE

Abstract—A typical approach to creating complex robot behaviors is to compose atomic controllers, or skills, such that the resulting behavior satisfies a high-level task; however, when a task cannot be accomplished with a given set of skills, it is difficult to know how to modify the skills to make the task possible. We present a method for combining symbolic repair with physical feasibility checking and implementation to automatically modify existing skills such that the robot can execute a previously infeasible task. We encode robot skills in linear temporal logic (LTL) formulas that capture both safety constraints and goals for reactive tasks. Furthermore, our encoding captures the full skill execution, as opposed to prior work where only the state of the world before and after the skill is executed are considered. Our repair algorithm suggests symbolic modifications, then attempts to physically implement the suggestions by modifying the original skills subject to Linear Temporal Logic (LTL) constraints derived from the symbolic repair. If skills are not physically possible, we automatically provide additional constraints for the symbolic repair. We demonstrate our approach with a Baxter and a Clearpath Jackal.

Index Terms—Failure detection and recovery, formal methods in robotics and automation, specification repair, task planning.

I. INTRODUCTION

A CENTRAL challenge in robot planning and control is how to automatically and correctly generate robot behavior that satisfies complex, high-level tasks, especially those where the robot must react to events in the environment (i.e., reactive tasks). One common approach is to compose skills—atomic behaviors—that are simpler to define and verify than the full complex behavior. For example, for a table setting robot, it is simpler to provide the robot with individual skills for placing silverware, plates, and cups, and define tasks for performing different place settings, than to create one “set table” skill.

There are multiple approaches to skill composition, from chaining skills/controllers together [1], [2], [3], [4] to composing them based on symbolic controllers synthesized from temporal logic [5]. Typically, such approaches abstract skills and states as symbols, then synthesize a symbolic high-level strategy that is used to govern the execution of the skills by the physical system, based on environment or user-provided input. While powerful, such approaches traditionally make two assumptions that may not hold; first, that skills are abstracted by defining preconditions and postconditions (e.g., [1], [6], [7])—states in which the skill can be executed and states that are the result of applying the skill, respectively—while in fact, during skill execution the system might be temporarily in another state. Second that all the skills needed to achieve the task are given. In this work we relax those two assumptions.

For the first assumption, without considering what happens during the execution of the skill, the tasks we can specify and verify are limited—we cannot verify safety constraints or goals that might be reached during the execution of the skill. For example, we might not want the table setting robot to move drinks over any electronics on the table, but given only the start location of the drink and the end location (state before and after the skill execution), we might think the constraint is satisfied, while in reality it is not. In another task for the same robot, we might not want to move drinks over plates when they have food on them, but are unconcerned about moving drinks over electronics. Essentially, we may have task-dependent constraints that we would not want to enforce at the low-level skill controller. In this work we explicitly consider such intermediate states, thereby ensuring correct physical execution of the task. By considering intermediate states reached during the execution of the high-level task, we allow the user to define their desired specification and the system determines if the set of skills is sufficient for the task.

To relax the second assumption, ideally the system would be able to recognize and repair tasks that cannot be achieved with the set of skills the robot currently has. There exist several methods to repair temporal logic specifications (e.g., [8], [9], [10], [11], [12], [13], [14]), provide reasons for infeasibility (e.g., [15]), or learn skills subject to constraints (e.g., [16], [17], [18]). However, most methods that repair temporal logic specifications modify or relax the task assigned to the robot rather than modifying the robot’s capabilities [9], [10], [11], [12], [13], [14]. There is little work on automatically finding physically implementable skills to repair a high-level task without modifying the original task [8], [27]. In this work, we integrate symbolic repair of high-level tasks with physical feasibility checking and implementation of skills that were symbolically suggested.

In this work, we are given a reactive task encoded in the Generalized Reactivity (1) (GR(1)) fragment of linear temporal logic (LTL) [19], a set of skills, and an abstraction of the world.
We encode the skills in a GR(1) formula such that intermediate states are explicitly encoded. Given the task, we automatically find a strategy that enables the robot to complete it, if possible, or repair the task by finding and physically implementing skills that enable accomplishing the task.

During the repair process, not all skill suggestions will be physically possible. For example, they might require the robot to teleport between disjoint spaces or reach areas that the robot cannot reach. We therefore divide the task repair process into two parts: symbolic repair—finding, symbolically, suggestions for skill modifications—and physical feasibility checking and implementation—finding a controller able to physically implement the modification.

Crucial and unique to our approach is the feedback between the two parts; the symbolic repair provides suggestions to the physical repair, and the physical repair includes runtime monitors that, if a skill’s repair is not physically feasible, provide additional constraints to the symbolic repair. In addition, we allow the user to provide constraints on how skills can be repaired, for example restricting the symbols that are allowed to change; these (optional) constraints can be used to guide the repair toward more physically desirable modifications. Our repair process iterates between symbolic repair and physical implementation until a feasible physical skill, or set of skills, that repairs the overall task, is found.

Contributions: In this work, we provide a novel automated method for defining and repairing high-level, reactive behaviors given in GR(1) such that they are physically feasible. Our main contributions are as follows.

1) Skill encoding that enables synthesizing control with physically realistic safety constraints.
2) An automated process for physically implementing symbolic repair suggestions where the symbolic and physical repair are tightly integrated.
3) Demonstrations of the approach with two different physical robots, abstractions, and tasks.

II. RELATED WORK

A. LTL and Specification Repair

LTL has been used for specifying complex tasks in a variety of applications [5]. However, if a task cannot be completed as specified, it is difficult to determine the cause, much less what must be done to make the task possible. Recent work has investigated methods for debugging specifications that cannot be completed, leaving the user to determine the appropriate fix (e.g., [15], [20]). In [15], the authors proposed a method to highlight portions of an LTL specification that make the task not possible. Given a GR(1) specification that cannot be satisfied, [20] provides a countertrace that shows how the specification can be violated. Researchers have investigated the automatic repair of specifications; these methods often suggest adding restrictions to the environment to prohibit behaviors that would make the specification unrealizable (e.g., [9], [10], [21], [22], [23], [24]). Related to specification repair is the partial satisfaction of specifications, where a task is divided into hard constraints that must always hold and soft constraints that the robot does its best to satisfy (e.g., [25], [26]). The repair proposed in these works result in the robot completing a modified task, not the original task. Here, in contrast, we explicitly focus on modifying the capabilities of the robot to complete the original task.

This article builds on recent work on specification repair that modifies the skills of the robot rather than restrict the environment [8], [27]. This work, in contrast to [27] and [8], explicitly considers the intermediate states of skills, thereby we are able to capture the full physical execution, not only the preconditions and postconditions as the previous works did. There, multiple abstract new skills or modifications to skills were automatically suggested and then the user could choose which one(s) to try to physically implement on their own. This work integrates the symbolic and physical repair by automatically implementing the suggested modification and provides additional constraints to the repair process if the symbolic suggestions are not physically possible. The symbolic repair portion of this work additionally allows an entire new class of specifications to be repaired that [27] and [8] cannot, such as modifying the intermediate states of the table setting robot that moves drinks over electronics.

B. Relation to LTL Constraints for Physical Skills

Recent work has investigated incorporating LTL specifications into physical skills. Several works (e.g., [18], [28], [29], [30]) have investigated performing reinforcement learning subject to various types of LTL and symbolic constraints. Work has also investigated adding constraints to demonstrated skills (e.g., [16], [17]).

These results are complementary to this article—here we use the framework in [16] to create skills that satisfy our suggested LTL constraints. One can substitute other techniques for the physical repair.

C. Relation to Task and Motion Planning (TAMP)

Related to this work is the field of TAMP (for a review see [31]). In TAMP, actions or skills are often defined symbolically using a STRIPS-style framework [7], where each skill has a set of preconditions and effects. Skills in this work also have preconditions and postconditions, but also contain information on what occurs during the execution of the skill at the symbolic level. In this work, tasks are specified as LTL formulas that may include sequencing, conditions, and reactions to environment events, while tasks in TAMP are often to achieve a goal state or set. Given a task, TAMP finds a solution by typically combining symbolic search and geometric motion planners. In contrast, our approach assumes a set of skills that can be composed is given, and we check before execution if the skills can solve the task, and if not, perform repair. In general, TAMP is better suited for tasks that require motion and manipulation in highly cluttered environments (e.g., [32], [33], [34], [35]), while the approach proposed here is better suited for tasks that can be performed with a small curated set of skills but that include complex, temporally extended goals, reactions to environment events, and conditional safety requirements. For example, a robot stocking
inventory in a grocery store while working around people, a robot working in a hospital supporting nursing staff by delivering items, or a robot waiter reacting to requests to pour drinks or serve food from diners. The curated set of skills in the case of the robot waiter might include a skill to pour water, perhaps given via learning from demonstration [36].

Furthermore, while in TAMP the premise is that there exist (typically several) symbolic plans that can cause the system to reach its goal and the challenge is to find a physical implementation, in this work we repair the symbolic plan—the synthesis failing means there is no symbolic plan able to satisfy the task—while ensuring physical feasibility.

III. PRELIMINARIES

In this section we provide background information regarding skills, their abstraction, the logic we use to capture specifications (LTL), and techniques for modifying skills.

Example 1 (Nine squares): Consider a 2-D environment where the state (x,y) is abstracted by nine squares, as shown in Fig. 1. In the following and throughout the text, we will use this simple example to illustrate different skills, tasks, and the repair process. The full definition of skill L2R in LTL and an example of a repair appear in the Appendix.

A. Skills and Skill Abstraction

The robot operates in a continuous state space \((x_1, \ldots, x_n) \in X \subseteq \mathbb{R}^n\) with a given set of skills \(A\). A skill \(s\) is associated with a controller that causes the robot to change the physical state of the world. A skill can only be applied from a specific set of states (the precondition) and results in a set of states (the postcondition). While a skill is being executed, it may pass through an intermediate set of physical states that are neither the precondition nor the postcondition. We assume that a skill is an atomic action, that is, once it starts executing, it does not stop until the state is in the postcondition. In Example 1, shown as in Fig. 1, skill L2R’s precondition is \(0 \leq x < 1\) and \(0 \leq y < 1\), and its postcondition is \(2 \leq x < 3\) and \(2 \leq y < 3\).

To enable symbolic reasoning and specifying high-level behaviors, we use a discrete abstraction—a set of Boolean propositions—to describe the state of the world that is then used to define tasks and perform symbolic repair. We use the grounding of the abstractions—the physical meaning of each proposition—for the physical repair. We define \(I_G\) as the set of propositions that capture the state of the world. Each proposition \(\pi \in I_G\) grounds to a subset of the state space via the grounding operator \(G\) (i.e. \(G(\pi) \subseteq X\)). We define a symbolic world state, \(\sigma \in 2^h\), as a set of propositions, with the corresponding grounding \(G(\sigma) = \bigcap_{\pi \in \sigma} G(\pi)\). In Example 1, \(I_G = \{x_0, x_1, x_2, y_0, y_1, y_2\}\) and \(\sigma = \{x_0, y_0\}\) grounds to \(G(\{x_0, y_0\}) = \{0, 1\} \times \{0, 1\}\).

The initial precondition, the set of symbolic world states from which a skill \(s \in A\) can be executed, is given by \(\Sigma_{\text{pre-init}}^G \subseteq \sigma I_G\). The final postcondition, the set of symbolic world states a skill \(s \in A\) will result in, is given by \(\Sigma_{\text{post-final}}^G \subseteq \sigma I_G\). The propositions \(I_G\), and the sets \(\Sigma_{\text{pre-init}}^G\) and \(\Sigma_{\text{post-final}}^G\) can all be automatically created (using, e.g., [11]); in this work, we assume they are given.

In Example 1, \(\Sigma_{\text{pre-init}}^L2R = \{\{x_0, y_0\}\}\) and \(\Sigma_{\text{post-final}}^L2R = \{\{x_2, y_2\}\}\). In general, the initial preconditions and final postconditions can consist of multiple symbolic world states. For example, there could be a skill with a nondeterministic outcome \(\Sigma_{\text{post-final}} = \{\{x_2, y_0\}, \{x_1, y_1\}\}\), as in following examples.

B. Linear Temporal Logic

Formulas in LTL are written over a set of atomic propositions, \(\mathcal{V}\), where \(\pi \in \mathcal{V}\) is a Boolean proposition. The syntax of LTL is given by the following grammar:

\[
\varphi ::= \pi \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \mathcal{U} \varphi_2
\]

where \(\varphi\) is an LTL formula, \(\neg\) is negation, \(\lor\) is disjunction, \(\bigcirc\) is the next operator, and \(\mathcal{U}\) is the until operator. We define \(\text{True} \equiv \varphi \lor \neg \varphi\) and \(\text{False} \equiv \neg \text{True}\). Given these operators, one can derive conjunction \((\varphi_1 \land \varphi_2 \equiv \neg (\neg \varphi_1 \lor \neg \varphi_2))\), implication \((\varphi_1 \rightarrow \varphi_2 \equiv \neg \varphi_1 \lor \varphi_2)\), equivalence \((\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1))\), eventually \((\bigcirc \varphi \equiv \text{True} \mathcal{U} \varphi)\), and always \((\square \varphi \equiv \neg \bigcirc \neg \varphi)\).

We use prime (’) to denote propositions at the next time step (i.e., \(\varphi')\) with \(\mathcal{V}' = \{\varphi' \mid \varphi \in \mathcal{V}\}\).

The semantics of an LTL formula \(\varphi\) are defined over infinite words \(\omega = \sigma_0\sigma_1\sigma_2\ldots\) where \(\sigma_i \in 2^\mathcal{V}\) is the set of propositions that are True in step \(i\) [37]. We denote that a word \(\omega\) satisfies a formula \(\varphi\) at step \(i\) by \(\omega, i \models \varphi\). Intuitively, \(\omega, i \models \bigcirc \varphi\) if \(\varphi\) is True at step \(i + 1\), \(\omega, i \models \square \varphi\) if \(\varphi\) is True at some step after including \(i\), and \(\omega, i \models \square \varphi\) if \(\varphi\) is True at all steps after and including \(i\).

In this work, we consider the GR(1) fragment of LTL [19]. The set of propositions \(\mathcal{V}\) is partitioned into a set of input \((I)\) and output \((O)\) propositions \((\mathcal{V} = I \cup O)\). Input propositions \(I\) correspond to the state of the world, which ground to the physical world the robot can control through its skills \(I_G\) or that are controlled by the user \((\mathcal{E})\). User controlled propositions could correspond to user inputs, such as pressing a button or interacting with a GUI. The output propositions \(O\) are controlled by the system (or robot, which we use interchangeably throughout). In this work, they correspond to the activation of the skills the robot has; each skill \(s \in A\) has a corresponding \(\pi_s \in O\). In Fig. 3(a), \(I = \{x_0, x_1, x_2, y_0, y_1, y_2\}\) and \(O = \{\pi_{L2R}, \pi_{L2R'}\}\).

In GR(1), specifications take the form [19]

\[
\varphi = \varphi_c \rightarrow \varphi_s
\]
where $\phi_s = \phi_s^l \land \phi_s^v \land \phi_s^t$ contains assumptions about the environment behavior and $\phi_a = \phi_a^l \land \phi_a^v \land \phi_a^t$ contains requirements on the system’s behavior and:

- $\phi_s^l$ and $\phi_s^v$ are Boolean formulas over $I$ and $I \cup O$, respectively, characterizing the initial states;
- $\phi_s^l$ and $\phi_s^v$ are safety constraints of the form $\bigwedge_{i} \Box \psi_i$ where $\psi_i$ are over $v$ and $C \cup u$. For $\phi_s^l$, $v \in I \cup O$ and $u \in I$. For $\phi_s^v$, $v \in I \cup O$;
- $\phi_a^l$ and $\phi_a^v$ are the fairness assumptions and liveness requirements, respectively, that capture events that should occur repeatedly. Here, $\phi_a^l = \bigwedge_{i=1}^{n} \Box \Diamond J_i^l$ and $\phi_a^v = \bigwedge_{j=1}^{n} \Diamond \Box J_j^v$ where $J_i^l$ and $J_j^v$ are Boolean formulas over $I \cup O$.

C. Notation: Symbolic States and Corresponding Formulas

Given the discrete abstraction of the world and the skills, we define a symbolic state $\sigma$ as a set of propositions that are $\text{True}$. We use $\Sigma_\alpha = 2^{\omega}$ to denote a set of symbolic states and $\sigma_\alpha \in \Sigma_\alpha$ to denote a single symbolic state where $\alpha \in \{ I, I', O, O', V, V', I_G \}$. Note that we drop the subscript $\alpha$ from the symbolic state $\sigma_\alpha$ when it is clear which set of symbols it is drawn from. During the skill encoding process and for feedback from the physical implementation, we only use symbolic world states $\sigma \in 2^{I_G}$. During the repair process, we consider all symbolic states.

To represent a symbolic world state, we use both $\sigma$ and its corresponding Boolean formula $\phi_\sigma = \bigwedge_{\pi \in \sigma} \pi \bigwedge_{\pi \in \sigma} \neg \pi$. The formula evaluates to $\text{True}$ when the world is in symbolic state $\sigma$. We use both representations to simplify the descriptions in the article. We use the formulas $\phi_\sigma$ to encode skills in GR(1) and the feedback between the symbolic repair and physical feasibility. We use $\sigma$ when describing the repair process.

D. Synthesis

We use GR(1) synthesis [19] to find a strategy, a symbolic composition of skills, to accomplish a task. Synthesis involves solving a two-player game played between the system (robot) and its environment, where the environment plays first and the system reacts to it. Synthesis, as done in [19], is a worst-case approach—the environment is considered to be adversarial and it attempts to keep the system from accomplishing its task. This ensures that if synthesis returns a strategy, the system is guaranteed to accomplish its task regardless of what the environment does, including changes to user controlled propositions.

Following [19], we define a game structure as $G = (V, I, O, \theta_{\text{init}}, \tau_v, \tau_s, \Phi)$ where $V$ is the set of propositions and $I$ and $O$ are defined, as in Section III-B. We define $\theta_{\text{init}}$ as...
the set of states that satisfy $\varphi^g_i \land \varphi^d_i$. We define $\tau_c \subseteq \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{V}'}$ as the set of current states and next input states satisfying $\varphi^g_i$, and $\tau_s \subseteq \Sigma_{\mathcal{V}} \times \Sigma_{\mathcal{V}'}$ as the set of current and next states satisfying $\varphi^d_i$. Note that in [19], $\tau_c$ and $\tau_s$ are defined as logical formulas; here we define them as sets of sets of propositions. The winning condition is given by $\Phi = \varphi^g \rightarrow \varphi^d$.

Given a game structure $\mathcal{G}$, the realizability problem is to decide if the game is winning for the system; either (a) for every environment action, the system is able to achieve $\varphi_s$ or (b) the system is able to falsify $\varphi_e$. To determine if a specification is realizable, we find, via a fixed point computation, all the states $Z \subseteq \Sigma_{\mathcal{V}}$ from which the system is able to win [19]. We iterate through every system liveness guarantee, $J^s_i$, and determine the set of states the system can always either transition to the next liveness goal from or falsify $\varphi_e$. The synthesis problem is to compute a strategy for the system to make the specification realizable [19].

E. Physically Implementing Skills

There are many ways to physically implement skills for a robot (e.g., [16], [17], [28]). In this work, we use [16] for the physical feasibility checking and implementation of modified skills. In [16], given a set of demonstrations (trajectories), the authors transform LTL constraints into a loss function and use that to generate new trajectories that remain close to the original demonstrated trajectories while satisfying the LTL constraints, inspired by the robustness measure from temporal logic [16]. An LTL formula $\varphi$ is evaluated at a time step $i$ in a trajectory generated by a skill. For example, the loss function corresponding to $x_0$ evaluating to $\text{True}$ (where $\mathcal{G}(x_0) = [0, 1]$) in Fig. 1 at time step $i$ is $L_c(x_0 = \text{True}) = \max(\tau_c(0 \leq x, i), \tau_c(x < 1, 1))$. Here, $\max(\tau(x_0, \ldots, x_n) = \gamma \ln \sum_{i=0}^{\infty} \exp(x_i^\gamma)$ and $\tau_c(0 \leq x, i) = \max(0 - x(i), 0)$ and $x(i)$ is the $x$ value of the trajectory at time step $i$ of the trajectory execution. The authors combine the loss term corresponding to the full LTL formula $L_c(\varphi, 0)$ with another loss term $L_d$, corresponding to how close the new trajectory remains to the original trajectory. Weighting the terms $L_c$ and $L_d$ allows a user to tradeoff between satisfying the LTL specification and remaining close to the original trajectory [16].

Specifically, [16] models a skill as a Dynamic Motion Primitive (DMP) [38], which is a series of differential equations that use radial basis functions to express nonlinear trajectories. For a given skill, the start and end point are supplied to the DMP and the full discretized trajectory of the skill in $X$ is given. [16] train a feed-forward neural network to generate the weights associated with the radial basis functions using the LTL loss function and use these DMPs to generate trajectories, or more precisely, a series of waypoints, that satisfy the provided LTL constraints.

IV. Problem Statement

Given a robot with a set of skills ($\mathcal{A}$) represented using output propositions ($\mathcal{O}$), a set of input propositions ($\mathcal{I} = \mathcal{I}_0 \cup \mathcal{I}_c$), and a user provided (reactive) task $\varphi$, our goal is to find a set of skills and a composition strategy such that the robot can physically accomplish the user provided task.

We split this into two problems, (1) the encoding of the skills in a $\text{GR}(1)$ formula and synthesis, and (2) the repair process (symbolic and physical) when a task is unrealizable, given the skills. For the second problem, we allow the user to optionally provide constraints on the repair of the task, as discussed in the following.

Problem 1: Given a robot with a set of skills $\mathcal{A}$, encode the skills in a $\text{GR}(1)$ formula $\varphi$ such that safety constraints can be enforced on the intermediate states of skills and liveness can be achieved during the execution of skills.

Problem 2: Given a $\text{GR}(1)$ formula $\varphi$ encoding the skills $\mathcal{A}$ of a robot and the task, the controllers implementing the skills, the proposition groundings (G), constraints on which propositions can change, and physically impossible transitions, find physically implementable skills that result in the robot being able to achieve $\varphi$.

V. Approach

We present our approach to solving Problems 1 and 2 in Fig. 2. First, we encode the task and skills as a $\text{GR}(1)$ formula (see Section VI). If the specification is realizable, as in prior work, we create a strategy that the robot can execute (see Section III-B). If it is not realizable, we perform symbolic repair and generate symbolic suggestions for skills that respect both user provided and automatically generated constraints (see Section VII). Next, we attempt to physically implement the suggestions. If we are successful in physically implementing a skill, we encode it in the specification (see Section VII-D). We then perform synthesis again with the additional skill to check whether the task is realizable, and if it is not, we continue the repair process and find more skills to repair the specification (see Section III-B).

Two things to note: 1) the repair is performed offline; after the system is repaired and a strategy is generated, we can guarantee the robot will accomplish the desired task, and 2) the repair process only adds additional skills, it does not remove any of the original skills. By only adding skills, we ensure the capabilities of the robot only increase. By virtue of the synthesis process guaranteeing task completion, the robot will not execute a skill that would violate the task. If a skill should be removed, (e.g., if a mechanical defect makes the skill unsafe) the user can forbid it from being executed.

VI. Encoding Skills as GR(1) Formulas

In this work we are given a set of skills the robot has and we assume all related propositions and their groundings are given as well; either determined automatically (using, e.g., [1]) or provided by the user. Prior work [27] has shown how skills can be automatically encoded into GR(1) formulas and used to synthesize robot behaviors. In contrast to that work, here, in addition to capturing the preconditions and postconditions on a skill, we also explicitly consider how the state of the world changes while (after repair) the robot executes a skill. This captures the noninstantaneous nature of robot control and allows the user to receive safety guarantees on the full execution of the robot. Furthermore, instead of being required to satisfy a liveness
guarantee at the end of a skill, it can be satisfied during the skill execution. This allows skills to encode complete motions, instead of being artificially broken in the middle. For example, if a robot waiter needs to wipe a region of a table, the region could be an intermediate state rather than a postcondition.

We define $\Sigma_{\text{skill}} \subseteq 2^I$ as the set of unique symbolic world states visited during the execution of skill 1. From a given symbolic world state, $\sigma$, we define all possible next states when executing skill 1 as $\Sigma_{\text{skill}}^{\text{post}}, \sigma \subseteq 2^I$. Similarly, we define all the possible symbolic world states from which $\sigma$ could be reached by executing skill 1 as $\Sigma_{\text{skill}}^{\text{pre}}, \sigma \subseteq 2^I$. For convenience, we also define the set of states that are only intermediate states, i.e., not initial preconditions or final postconditions, as $\Sigma_{\text{skill}}^{\text{only-intermediate}} = \Sigma_{\text{skill}}^{\text{unique}} \setminus (\Sigma_{\text{skill}}^{\text{pre-init, } \pi} \cup \Sigma_{\text{skill}}^{\text{post-final, } \pi})$. In Fig. 1, for example, $\Sigma_{\text{skill}}^{\text{post,} \pi} = \{\{x_0, y_0\}, \{x_1, y_0\}, \{x_2, y_0\}, \{x_2, y_1\}, \{x_2, y_2\}\}$.

In this work, we do not assume these intermediate states are given—rather we determine $\Sigma_{\text{skill, } \pi}^{\text{pre}}, \sigma$ and $\Sigma_{\text{skill, } \pi}^{\text{post}}, \sigma$ by performing multiple physical executions of skill 1 and monitoring the result.

We automatically encode the skills in the GR(1) formula as part of $\varphi_\pi$ and $\varphi_s$. For the purposes of repair, we define the environment and system safety formulas as $\varphi_e = \varphi_e^{\text{mutable}} \land \varphi_e^{\text{hard}}$ and $\varphi_s = \varphi_s^{\text{mutable}} \land \varphi_s^{\text{hard}}$, respectively. The repair process may modify $\varphi_e^{\text{mutable}}$ (which captures the (possibly nondeterministic) postconditions, or outcomes, of skills) and $\varphi_s^{\text{mutable}}$ (which captures the preconditions of skills). We encode safety formulas that may not change and the repair process cannot modify, for example mutual exclusion of propositions that are grounded to disjoint sets (i.e., $G(\pi_1) \cap G(\pi_2) = \emptyset$). For example, for the abstraction in Fig. 3, $\square \neg(x_0 \land x_1)$ is part of $\varphi_e^{\text{hard}}$. We also require all propositions in $I_E$ to remain unchanged when no skill is active, i.e., $\Box((\bigwedge_{\pi \in O} \neg \pi_G) \rightarrow (\bigwedge_{\pi \in I_E} \pi_L \leftrightarrow \Box \pi_G))$.

### 2) Encoding Constraints on Skills in $\varphi_s^{\text{mutable}}$:

We encode preconditions and requirements stemming from intermediate states in $\varphi_s^{\text{mutable}} = \varphi_s^{\text{choose}} \land \varphi_s^{\text{continue}}$, as shown in (3) shown at the bottom of this page and (4) shown at the bottom of this page. The formula $\varphi_s^{\text{choose}}$ restricts when a skill can be activated; it prevents the system from choosing a skill when: 1) none of the initial preconditions are satisfied, and 2) the skill is not in the middle of being executed (i.e., the system is not in an intermediate state of the skill). This, therefore, allows the system to choose a skill only when a precondition is satisfied, but does not require the system to do so. The formula $\varphi_s^{\text{continue}}$ requires the system to continue executing a skill from intermediate states it reaches during execution, i.e., a skill cannot be aborted in the middle of the execution. We provide an example of $\varphi_s^{\text{mutable}}$ for skill L2R in (9) in the Appendix.

### 3) Hard Constraints:

As described previously, certain safety constraints may not be changed by the repair process. We encode these in $\varphi_e^{\text{hard}}$ and $\varphi_s^{\text{hard}}$. In $\varphi_s^{\text{hard}}$, we require that at most one skill is active at any given time. In $\varphi_e^{\text{hard}}$, we encode mutual exclusion of propositions that are grounded to disjoint sets (i.e., $G(\pi_1) \cap G(\pi_2) = \emptyset$). For example, for the abstraction in Fig. 3, $\square \neg(x_0 \land x_1)$ is part of $\varphi_e^{\text{hard}}$. We also require all propositions in $I_E$ to remain unchanged when no skill is active, i.e., $\Box((\bigwedge_{\pi \in O} \neg \pi_G) \rightarrow (\bigwedge_{\pi \in I_E} \pi_L \leftrightarrow \Box \pi_G))$.

### VII. SYMBOLIC REPAIR

When a specification is unrealizable, we perform symbolic repair, which returns symbolic suggestions for skill modifications that would allow the task to be completed. We leverage the general framework presented in [8] and propose different methods to modify the precondition and postconditions that take into account the assumptions we have on intermediate states. We give a brief summary of the overall framework; for further details see [8].

When performing synthesis for the full GR(1) specification, we detect the specification is no longer realizable when the current set of winning states, from which the robot is able to achieve all of the liveness goals, no longer overlaps with a set corresponding to a liveness goal or the initial conditions. The currently winning set and current liveness guarantee or initial set of states are passed to the repair process, along with sets corresponding to the preconditions, intermediate states, postconditions, and restrictions on which skills can be modified. During the repair process, our algorithm attempts to add or
Algorithm 1: Modify Preconditions.

**Input:** $\tau_e, \tau^\text{hard}_e, \tau^\text{mutable}_e, \tau^\text{new}_e, Z, I_0$

**Output:** $\tau^\text{new}_e, \tau^\text{mutable}_e$

// States where the system is guaranteed to reach Z for any environment choice
1 $T^\text{always-win-and-reachable} = \text{systemAlwaysWinsAndSkillsAreReachable}(\tau^\text{mutable}_e, \tau_e, Z)$;

// Find allowed changes to preconditions
2 $T^\text{allowable-changes} = \text{findAllowableChanges}(T^\text{always-win-and-reachable}, T^\text{hard}_e, T^\text{mutate-transitions}_e, T^\text{post}_e, \tau_e)$;

// Randomly select the possible change to add
3 if $T^\text{allowable-changes} \neq \emptyset$ then
4 $T^\text{selected-change} = \text{randomlySelectChange}(T^\text{allowable-changes}, T^\text{mutable}_e)$;
5 $T^\text{selected-change} = \text{removeDependenceOnUserPropositions}(T^\text{selected-change}, I_0)$;
6 else
7 return $T_e, \tau^\text{mutable}_e$;
8 end
9 $T^\text{new-skills} = \text{replacePreconditions}(T^\text{selected-change})$;

// Add transitions to $\tau^\text{mutable}_e$
10 $T^\text{new-pre-condition} = \text{addPreCondition}(T^\text{selected-change}, \tau^\text{mutable}_e)$;

// Determine if this is the initial precondition
11 if isInitialPrecondition($T^\text{new-pre-condition}$) then
12 $T^\text{new-pre-condition} = \text{disallowExecutionFromOtherIntermediateStates}(\tau^\text{mutable}_e, T^\text{new-pre-condition})$;
13 end
14 $\tau^\text{mutable}_e = \tau^\text{mutable}_e \cup T^\text{new-pre-condition} \cup T^\text{new-skills}$;

// Add transitions to $\tau^\text{mutable}_e$
15 $\tau^\text{mutable}_e = \text{createNewEnvTransition}(\tau_e, \tau^\text{hard}_e, T^\text{new-skills}, T^\text{selected-change}, \tau^\text{mutable}_e)$;

16 return $\tau^\text{new}_e, \tau^\text{mutable}_e$

modify transitions to connect the winning set with the previously unreachable liveness goal or initial set. Once we have modified the transitions to expand the winning set, we restart the synthesis process. We keep performing synthesis and then repair until either the specification is realizable or no further modifications can be found, in which case the repair cannot be done.

Similar to [8], we alternate modifying the preconditions and the postconditions of skills; we either relax preconditions to make skills applicable from more states, or modify postconditions that can reduce a skill’s nondeterminism (and thereby not allow the skill to enter unwanted states). However, in contrast to [8], here we:
1) allow the repair process to modify the intermediate states;
2) always modify existing skills (i.e., we do not create new skills);
3) allow the user and/or the physical repair to enforce constraints on the symbolic repair.

We present the algorithm for modifying the preconditions first (see Section VII-B) and then the algorithm for modifying the postconditions (see Section VII-C). However, when performing repair, we also generate suggestions by modifying the postconditions first and then the preconditions, as the resulting suggestions may be different. We use Example 1 to illustrate the repair process. Here, we are given two skills (L2R and R2L), as shown in Fig. 3(a). The user-specified task is “repeatedly visit the upper right and lower left corners ($\varphi^*_e = \square (x_2 \land y_2) \land \Box \neg (x_0 \land y_0)$) while always avoiding the lower right corner ($\varphi^*_\text{user} = \square (x_2 \land y_0) \land \Box \neg (x_2 \land y_0)$) starting from the lower left corner ($\varphi^*_i = x_0 \land y_0$), as shown in Fig. 3(b).

A. Constraints

In this section we discuss constraints we impose on the repair process. We describe constraints on how input propositions can change (see Section VII-A1) and on modifications that are known to be physically impossible (see Section VII-A2).

1) Constraints on Changes to $I_0$: We allow the user to restrict, if they choose to, how symbolic world states $\sigma \in I_0$ (i.e., the truth values of the world input propositions $I_0$) are allowed to change in the repair process. We specify the allowed modifications as $\varphi^\text{poss-changes} = \Box \varphi_i$ where $\varphi_i$ is a Boolean formula over $\Gamma \cup \Box \Gamma$. This formula is only used in the repair process and is not part of the task when synthesizing. We define the set $\tau^\text{poss-changes} \subseteq \Sigma_\Gamma \times \Sigma_\Gamma$ as pairs of input states that satisfy $\varphi^\text{poss-changes}$. During the repair process, the preconditions (or postconditions) of a skill can be modified from $\sigma_1 \text{ to } \sigma_2$, as shown in Fig. 3(c).

We show an example of $\varphi^\text{poss-changes}$ in Fig. 3(c). Consider a skill that goes from $\{x_2, y_1\}$ to $\{x_2, y_2\}$. If $\varphi^\text{poss-changes} = \Box (x_2 \land \Box (x_0 \land y_2)) \land \Box (y_1 \land \Box (x_0 \land y_0))$, we are not allowed to change $x_2$ to either $x_0$ or $x_2$ and are also not allowed to change $y_1$ to $y_0$. This only allows us to modify the preconditions such that a candidate modified skill goes from $\{x_1, y_1\}$ to $\{x_2, y_2\}$ or from $\{x_1, y_2\}$ to $\{x_2, y_2\}$, as shown in Fig. 3(c).

2) Disallowed Transitions Due to Physical Infeasibility: We place constraints on which transitions can be suggested
for a given skill; we primarily use these constraints to automatically incorporate feedback from the physical implementation process where we find transitions that are not physically possible, although these constraints can be given by the user as well. We write these constraints as \( \varphi_{\text{disallowed-transitions}} = \land_i \Box \neg \varphi_i \) where \( \varphi_i \) is a Boolean formula over \( V \cup \bigodot I \). We create the set \( \tau_{\text{disallowed-transitions}} \subseteq \Sigma_V \times \Sigma_Z \) which contains pairs of states that satisfy \( \varphi_{\text{disallowed-transitions}} \). As with \( \tau_{\text{possible-changes}} \), \( \tau_{\text{disallowed-transitions}} \) is only used in the repair process, not the synthesis process. For example, if \( \varphi_{\text{disallowed-transitions}} = \Box \neg (x_1 \land y_1 \land \bigodot x_2 \land \bigodot y_2) \), the repair process cannot suggest a modified skill going from \( \{x_1, y_1\} \) to \( \{x_2, y_2\} \), as shown in Fig. 3(d).

### B. Modify Preconditions

Given a skill, the physical meaning of modifying its preconditions is allowing it to be executed from a different set of initial states, with the goal of making the task realizable. Given the current environment transitions (\( \tau_e \)), the current hard environment transitions (\( \tau_{e}^{\text{hard}} \)), the set of changeable system transitions (\( \tau_{c}^{\text{mutable}} \)), the set of hard system transitions (\( \tau_{c}^{\text{hard}} \)), and the set of winning states (\( Z \)), we find skills that always reach \( Z \). We then modify the preconditions of those “winning” skills, subject to \( \tau_{c}^{\text{mutable}} \) and \( \tau_{c}^{\text{hard}} \), to enable the system to execute these skills from more states; we add the modified preconditions to the existing preconditions of that skill in \( \tau_{s}^{\text{mutable}} \). We also modify \( \tau_{e}^{\text{hard}} \) to allow the new preconditions to reach the appropriate postcondition and if necessary be reached (if it is an intermediate state) by the modified skill. We present our algorithm for modifying preconditions in Algorithm 1 and in more detail in Algorithm 3 in the Appendix. Line numbers in this section reference Algorithm 1.

We find the set \( \tau_{\text{always-win-and-reachable}} \subseteq \Sigma_V \times \Sigma_Y \) (Line 1) such that \( \sigma' \in \Sigma_Y \) is a winning state (i.e., \( \sigma' \in Z \)) and \( \sigma \in \Sigma_V \) is reachable, i.e., there exists a transition in \( \tau_{e} \) that results in \( \sigma \).

In Example 1, the repair process is triggered when the system on the process attempts to reach \( \{x_2, y_2\} \). Here, \( Z = \{\{x_2, y_1\}, \{x_2, y_2\}, \{x_1, y_2\}, \{x_2, y_2\}\} \), as shown in Fig. 4(a). We find \( \tau_{\text{always-win-and-reachable}} = \{\{x_2, y_0, x_1, y_2, \pi_{L2R}, x_2', y_1', \pi_{R2L}\}, \{x_2, y_1, x_1, y_2, \pi_{L2R}, x_2', y_2', \pi_{R2L}\}\} \) [see Fig. 4(b)].

We then find allowable changes to the preconditions; we capture allowable changes by specifying current states \( \Sigma_Y \), next states \( \Sigma_Y' \) and possible new preconditions \( \Sigma_{X_1} \) in \( \tau_{\text{possible-changes}} \subseteq \Sigma_Y \times \Sigma_Y' \times \Sigma_{X_1} \) in Line 2. The changes to \( \tau_{\text{always-win-and-reachable}} \) are such that the new preconditions are allowed by the change constraints \( \tau_{\text{possible-changes}} \), which do not violate \( \tau_{\text{disallowed-transitions}} \), and satisfy the hard system constraints \( \tau_{c}^{\text{hard}} \).

We also require that the new preconditions are not an intermediate state already in the skill.

In Example 1, the allowable changes are shown in Fig. 4(c). The current preconditions, shown with double hashes, of \( \{\{x_2, y_0\}, \{x_2, y_1\}\} \) can be modified to be any of the states shown with vertical hashes, i.e., \( \{\{x_0, y_1\}, \{x_0, y_2\}, \{x_1, y_1\}, \{x_1, y_2\}\} \). As opposed to the example in Fig. 3(c), here we only require that the changes to the input propositions respect the mutual exclusion of propositions in \( \tau_{\text{possible-changes}} \) (e.g., \( \Box \neg \bigodot (x_0 \land x_1) \)). Requirements that the new preconditions are not already states of the skill additionally disallow states where \( y_0 \) or \( x_2 \) are True.

From these possible precondition modifications, we randomly choose a modification in Line 4. Here, we assume that the selected modification is \( \tau_{\text{selected-change}} = \{\{x_2, y_0, \pi_{L2R}, x_2', y_1', \pi_{R2L}, x_1', y_1\}\} \), as shown in Fig. 4(d). Note that \( \tau_{\text{selected-change}} \) may contain more than one modification. For
example, if we modified the precondition \( \{x_2, y_1\} \), \( \tau_{\text{selected-change}} \) would include both \( \{x_2, y_1, \pi_{\text{L2R}}, x_2', y_2', \pi_{\text{RL2}}, x_1, y_1\} \) and \( \{x_2, y_1, \pi_{\text{L2R}}, x_2', y_2', x_1, y_1\} \) (the difference being the skill or lack thereof in the next state).

Since we do not allow our modifications to restrict user-controlled propositions, in Line 5 we add all possible combinations of user controlled propositions \( \mathcal{X}_u \) if they exist, to the suggested change. Essentially, we are creating multiple copies of the modification, one for each unique truth assignment to the user controlled propositions, thus encoding that those propositions can take on any value.

We add the new (i.e., modified) skill \( (\tau_{\text{new-skill}}, \text{Line 9}) \) to \( \tau_{\text{mutable}} \) and \( \tau_e \). In Example 1, \( \tau_{\text{new-skill}} = \{x_1, y_1, \pi_{\text{L2R}}, x_2', y_2', \pi_{\text{RL2}}, x_1, y_1\} \) [see Fig. 4(d)].

In order to ensure that the new precondition is reachable (otherwise it cannot be executed and thus cannot repair the task), we determine the preconditions of the original precondition and allow the modified precondition to be executed from those states \( (\tau_{\text{new-pre-condition}}, \text{Line 10}) \). In Example 1, originally L2R goes from \( \{x_1, y_0\} \) to \( \{x_2, y_0\} \). For the modified L2R, we add \( \{x_1, y_0, \pi_{\text{L2R}}, x_1, y_0, \pi_{\text{RL2}}\} \) to \( \tau_{\text{mutable}} \) thereby allowing the skill to reach \( \{x_1, y_1\} \) from \( \{x_1, y_0\} \), as shown by the black headed arrow in Fig. 4(e). Two things to note here as as follows.

1) The added transitions might be physically impossible—they are not subject to \( \tau_{\text{disallowed-transitions}} \)—such modifications will be automatically rejected by the physical feasibility checking.

2) We are adding transitions but not removing any—essentially we are adding nondeterminism to the skill.

If the modified initial preconditions, \( \tau_{\text{new-pre-condition}} \), allows the skill to be executed from the new preconditions no matter the previous skill, then we add \( \tau_{\text{new-pre-condition}} \) to \( \tau_{\text{mutable}} \), this can override the requirement that a skill continue executing until it reaches its final postcondition. We therefore adjust \( \tau_{\text{new-pre-condition}} \) when the initial precondition is modified to prevent the strategy from switching skills midexecution. For example, if we attempt to modify the initial precondition \( \{x_0, y_0\} \) of L2R to \( \{x_0, y_1\} \), \( \tau_{\text{new-pre-condition}} = \bigcup_{x \in \Sigma_0} \sigma \cup \{x_0', y_1', \pi_{\text{RL2}}\} \).

When added to \( \tau_{\text{mutable}} \), this would allow the robot to execute skill L2R from \( \{x_0, y_1\} \) instead of being required to continue executing skill R2L until \( \{x_0, y_0\} \) is reached. We do not want this, so we remove the offending transitions from \( \tau_{\text{new-pre-condition}} \) in Line 12.

The new mutable system transitions are the original mutable transitions with the new preconditions and the preconditions that allow that precondition to be reached (Line 14).

For \( \tau_{\text{new}} \), we add the new precondition in \( \tau_{\text{new-skill}} \) and enforce that it reaches the original postcondition (Line 15). We also add the precondition and postcondition in \( \tau_{\text{new-pre-condition}} \). In Example 1, we add \( \{x_1, y_0, \pi_{\text{L2R}}, x_1', y_1'\} \) and \( \{x_1, y_1, \pi_{\text{L2R}}, x_2', y_2'\} \) to \( \tau_e \) and disallow \( \{x_1, y_1, \pi_{\text{RL2}}\} \bigcup_{\sigma \in \Sigma_0} \{x_2', y_2'\} \sigma \) in \( \tau_e \), essentially not allowing transitions from \( \{x_1, y_1\} \) to any state other than \( \{x_2, y_1\} \). We show the resulting new skill in Fig. 4(f).

C. Modify Postconditions

When modifying postconditions, we automatically change skills that do not lead to a winning state to have postconditions that result in states that are winning, subject to \( \varphi_{\text{disallowed-transitions}} \). This process might reduce nondeterminism (though not in all cases), and in contrast to [8], we do not restrict ourselves to only removing postconditions that are not winning, but also allow adding new ones. We show the process of modifying the postconditions in Algorithm 2 and in more detail in Algorithm 4 in the Appendix. In this section, all line references are to Algorithm 2. Fig. 5 shows the process of modifying the postconditions in Example 1 after modifying the preconditions, as described in Section VII-B and Fig. 4.

---

**Algorithm 2: Modify_Postconditions.**

**Input:** \( \tau_e, \tau_{\text{mutable}}, Z, I_0 \)

**Output:** \( \tau_{\text{new}} \)

// Find states that do not lead to winning states
1. \( \tau_{\text{skills}} = \text{findReachableSkills}(\tau_{\text{mutable}}, \tau_e) \);
2. \( \tau_{\text{not-winning}} = \text{findNotWinningSkills}(\tau_{\text{skills}}, Z) \);

// Find allowable changes
3. \( \tau_{\text{allowable-changes}} = \text{findAllowableChanges}(\tau_{\text{not-winning}}, \tau_{\text{post-changes}}, \tau_{\text{disallowed-transitions}}, \tau_{\text{mutable}}, \tau_e, Z) \);
4. if \( \tau_{\text{allowable-changes}} \neq \emptyset \) then
5. \( \tau_{\text{selected-change}} = \text{randomlySelectOneChange}(\tau_{\text{allowable-changes}}) \);
6. \( \tau_{\text{selected-change}} = \text{removeDependenceOnUserPropositions}(\tau_{\text{selected-change}}) \);
7. else
8. return \( \tau_e \)
9. end

// Modify \( \tau_e \)
10. \( \tau_{\text{env-add}} = \text{findEnvTransitionsToAdd}(\tau_{\text{selected-change}}) \);
11. \( \tau_{\text{env-old}} = \text{findEnvTransitionsToRemove}(\tau_{\text{selected-change}}) \);
12. \( \tau_{\text{new}} = (\tau_e \setminus \tau_{\text{env-old}}) \cup \tau_{\text{env-add}} \);
13. return \( \tau_{\text{new}} \)

---

**Fig. 5.** Modifying postconditions in Example 1 assuming the change in Fig. 4(d) has been selected. (a) New winning states Z when the system is trying to satisfy the liveness \( \square (x_2 \land y_2) \). The input states are the shaded squares and the arrows are the skill being applied. (b) Transitions that are currently not winning. The preconditions are shown by the shaded squares, the current skill with the thick arrow, and the postcondition with the double hashes. (c) Allowable changes to the postconditions are shown with vertical hashes. The arrow with the white head shows the possible new transition. (d) New skill after modifying the postconditions.
In Example 1, after modifying the preconditions, we attempt to synthesize a winning strategy, but the repair process is again triggered when trying to reach \( \{ x_2, y_2 \} \). Since we modified the preconditions, there are more states from which we can reach \( \{ x_2, y_2 \} \) and \( 2 = \{ x_2, y_1, \pi_{L2R}, \{ x_2, y_2, \pi_{L2R}, \{ x_1, y_1, \pi_{L2R} \} \} \), as shown in Fig. 5(a). From these states, we are either already in \( \{ x_2, y_2 \} \) or are guaranteed to reach it at some future point.

We first find all of the transitions whose preconditions are reachable (Line 1) and then identify those skills that neither start in nor result in \( Z(\tau_{\text{not-winning}}) \) and \( \tau_{\text{L2R}} \). Fig. 5(b) depicts transitions in \( \tau_{\text{not-winning}} \) that could result in task failure due to nondeterminism of the skill. The components of \( \tau_{\text{L2R}} \) that are not winning are:

\[
\begin{align*}
(5) & = \{ x_0, y_0, \pi_{L2R}, x_1, y_0 \}
\end{align*}
\]

For example, \( \{ x_0, y_0, \pi_{L2R}, x_1, y_0 \} \) results in \( \{ x_1, y_0 \} \) at the next step, and must continue executing \( \tau_{\text{L2R}} \), which may cause the system to reach \( \{ x_2, y_0, \pi_{L2R} \} \) (which violates the safety constraint and is thus not in the winning set \( Z \)).

Once we have all the transitions that are not winning, we determine which postconditions we can change so they are winning, and how to change them. By changing the postconditions of existing skills so they are winning, we will increase the set of winning states. The allowable changes are contained in \( \tau_{\text{allowable-changes}} \) (Line 3).

We include a transition in \( \tau_{\text{allowable-changes}} \) if it satisfies the constraints on skill modification specified by the user (\( \tau_{\text{possible-changes}} \)), is not disallowed (\( \tau_{\text{disallowed-transitions}} \)), is a system transition we can modify (\( \tau_{\text{mutable}} \)), does something (i.e., not a self-transition), and is winning (the postcondition is in the winning set). We only allow modifications that either reduce nondeterminism or result in postconditions that are not already states in the skill. From \( \tau_{\text{allowable-changes}} \), we randomly select one change (\( \tau_{\text{selected-change}} \), Line 5).

Fig. 5(c) shows an allowable change for \( \tau_{\text{L2R}} \); we can change the postcondition \( \{ x_2, y_0 \} \) to \( \{ x_1, y_1 \} \). Fig. 5(d) shows the resulting skill after making the modification in Fig. 5(c).

Similar to Algorithm 1, if there are user controlled prepositions, (i.e., \( 2 \neq \emptyset \)), we allow all possible combinations of user controlled variables in the selected modification (Line 6).

After selecting a change to make to the postconditions, we modify \( \tau_{\text{e}} \) by removing the transitions associated with the original postcondition and adding the transitions associated with the modified postcondition (Lines 10–12). If the modification in Fig. 5(d) is chosen, we remove the transition \( \{ x_1, y_0, \pi_{L2R}, x_2, y_0 \} \) and add the transition \( \{ x_1, y_0, \pi_{L2R}, x_1, y_1 \} \) to \( \tau_{\text{e}} \). Note that in this example, due to the precondition modification, \( \{ x_1, y_0, \pi_{L2R}, x_1, y_1 \} \) is already part of \( \tau_{\text{e}} \), so adding it has no effect and the meaningful change is removing a transition.

\[ \quad \]

D. Extracting a Strategy and Modifications to Skills

The symbolic repair may provide several skill modifications; it may be the case that not all are required for task completion. To physically implement only required new skills, when the repair is successful, i.e., the task is realizable given the original and new skills, we extract the required symbolic changes to the skills by finding a deterministic strategy and comparing \( \tau_{\text{e}} \) and \( \tau_{\text{mutable}} \) of the deterministic strategy to the original \( \tau_{\text{e}} \) and \( \tau_{\text{mutable}} \). This skill is then checked for physical feasibility.

For Example 1, when selecting the change in Fig. 5(d), we find we need a new skill based on \( \tau_{\text{L2R}} \) with intermediate states, as shown in (5). The LTL formulas (2)–(4) for a modified skill \( \tau_{\text{L2R}} \) in Fig. 5(d) are shown in (10) and (11) in the Appendix.

\[ \quad \]

VIII. PHYSICAL IMPLEMENTATION OF SKILLS

Once we have symbolic suggestions for skill modifications, we determine whether the skills are physically implementable. In this work we treat skills as trajectories generated by DMPs; however, other approaches would also work provided they can be modified subject to LTL constraints (e.g., [39], [40], [41]).

The first step is finding a new skill that has trajectories as close as possible to the original skill while satisfying the new constraints. If we find such a skill, we check whether the robot can physically follow the resulting trajectories. If, at either step, we find that the new skill is not possible, we determine which parts of the skill are not feasible and add constraints to the symbolic repair process, as described in Section IX.

There are multiple methods to find modifications to skills given symbolic constraints (e.g., [16], [17]). For this work, we use the method proposed in [16]; it transforms an LTL constraint into a loss function. Together with the initial skill, this constraint allows us to generate a new DMP.

To use the method in [16], we write each skill suggestion as an LTL formula in the form of conjunction of implications, as shown in (6). The formula requires skills to always go from a precondition to either the same precondition or one of the postconditions and always stay in the unique states of the skill. In the suggestion for Example 1

\[ \quad \]

Equation (12) in the Appendix encodes the new constraints on the skill

\[
\begin{align*}
\sigma_{\text{pre}} & \in \Sigma_{\text{unique}} \quad (12) \\
\psi_{\sigma_{\text{pre}}} & \to \bigwedge \left( \bigvee \psi_{\sigma_{\text{pre}}} \sigma_{\text{post}} \in \Sigma_{\text{skill}}, \sigma_{\text{pre}} \right)
\end{align*}
\]

To modify the skill, we take the original skill and the LTL constraint generated from the symbolic repair and train a neural network that generates the weights for a DMP that outputs the trajectory of the modified skill. The inputs to the skill modification process are starting and ending points sampled from the preconditions and postconditions of the skill, the original trajectories used to train the skill, and the constraints on the new skill (6). The neural network itself has a single hidden layer.
fully connected with ReLU activation, as in [16]. The method proposed by [16] allows the user to tune the tradeoff between remaining close to the original demonstrations versus obeying the LTL constraints. In this work, we weighted satisfying the LTL constraints 50 times more than remaining close to the original demonstrations. For each skill suggestion, we perform 100 epochs of training to create the neural network that generates the weights for the DMP. The second column of Fig. 6 shows the original trajectories for the skill L2R in Example 1. The third and fourth columns of Fig. 6(c) show the modified trajectories generated by the DMP subject to the LTL constraint generated from the suggestion in (5) and based on the original skill. The third column of Fig. 6(c) shows the trajectories that satisfy the constraint and the fourth column shows the trajectories that do not.

IX. FEEDBACK BETWEEN SYMBOLIC AND PHYSICAL REPAIR

Crucial and unique to our approach is the tight integration between the symbolic and physical repair. As described in the previous sections, the symbolic repair provides constraints to the physical repair; in this section we describe the constraints the physical repair provides the symbolic.

After attempting to physically implement the modified skill (see Section VIII), we check whether it satisfies the new constraints. We do this in two steps: first we gain confidence that the new skill satisfies the constraints, by sampling trajectories and ensuring the percentage of trajectories that satisfy the constraints (6) is above a threshold, here 90%. We randomly sample start and goal points and generate trajectories using the learned weights for the DMP. We verify that each of the trajectories satisfies the constraints by confirming that the loss function is zero for the generated trajectory.

We chose a threshold of 90% to indicate success of the skill; we would expect the repair algorithm to return skills that were not physically possible if we lowered this threshold. If we increased the threshold, we would need to tune the parameters for the learning process to make sure we did not falsely reject physically implementable skills.

Second, we verify that there exists a controller for the robot that can implement the trajectory. For example, for the Baxter (see Section XI-A), we check that the sampled end-effector trajectories have valid inverse kinematics solutions. If the new trajectory satisfies the constraints and a controller exists, the new skill is used to repair the task. For the (simulated) nine square example (Example 1), we assume a holonomic point robot that can reliably follow any trajectory, therefore we only consider the trajectory validity.

If a physically feasible trajectory was not found, we determine which transitions are physically unlikely and use those transitions to restrict future symbolic suggestions, by adding them to ϕ\text{disallowed-transitions}. Specifically, for each intermediate transition of the skill, we create a monitor that checks the formula in (7)—whether for each precondition, the trajectory either stays in the precondition or moves to an appropriate postcondition [based on (6)]

\[
\varphi_{\text{monitor}} = \square(\varphi_{\text{pre}} \rightarrow \bigcirc \left( \varphi_{\text{pre}} \lor \bigvee_{\pi_{\text{skill}} \in \varphi_{\text{post}}} \varphi_{\text{post}} \right)).
\]

For each transition that did not robustly succeed, we add \(\neg(\varphi_{\text{pre}} \land \varphi_{\text{skill}} \land \bigcirc \bigvee_{\pi_{\text{pre}} \in \varphi_{\text{post}}} \varphi_{\text{post}})\) to \(\varphi_{\text{disallowed-transitions}}\). This does not allow the repair to suggest a modification to \(\pi_{\text{skill}}\) that has a transition with preconditions \(\pi_{\text{pre}}\) and postconditions \(\pi_{\text{post}} \in \pi_{\text{skill}}\). Note that this transition is also disallowed for intermediate states. We show the integrated symbolic-physical repair process in Fig. 6. The first two rows show physically impossible symbolic suggestions, the attempted physical implementation, and the additions to \(\varphi_{\text{disallowed-transitions}}\). After forbidding those transitions, we get a physically possible symbolic suggestion, as shown in the third row of Fig. 6.

X. SIMULATED DEMONSTRATIONS

We present several demonstrations showcasing the different capabilities of the repair process, including multiple skills, modifying intermediate states, preconditions, postconditions, and achieving liveness at both the postcondition and during the skill. The code for the algorithms is available at.\(^2\) Fig. 7 illustrates the examples and shows the skills and tasks, together with symbolic suggestions that were found to be physically impossible (squares with a dashed outline and red “x”), and physically implementable skills that resulted from the repair (squares with a bold outline). Each bold square represents one set of suggestions from the repair process, which may include changes to multiple skills. Here, each trajectory is modeled as a DMP with 50 discrete points.

A) Modify Intermediate States: For Example 1 (\(\varphi_{\text{pre}} = \square(\chi_0 \land y_0) \land \square(\chi_2 \land y_2)\)) and \(\varphi_{\text{post}}^{\text{onest}} = \square(\chi_2 \land y_0) \land \square(\chi_0 \land y_0) \land \square(\chi_2 \land y_2)\)

\(^{2}\)[Online]. Available: https://github.com/apacheck/synthesis_based_repair
in Fig. 7(a), one repair is depicted in Fig. 6, and another in Fig. 7(a).

B) Modify Two Skills: Fig. 7(b) shows the same task as the previous one, but with three skills instead of two. The repair process suggests physically implementable modifications to two skills. These suggestions modify the final postconditions for one skill and the initial preconditions for the other.

C) Modify Two Skills Intermediate States: Fig. 7(c) shows an example where the hard constraint is to avoid \( \{ (x_1, y_1) \} \) and the liveness is the same as in Example 1. The repair process finds suggestions that modify the intermediate states, eliminating nondeterminism.

D) Modify Postconditions: In Fig. 7(d), the repair process suggests modifying the postconditions of the skill with nondeterministic outcomes. Note that the skill with a precondition that includes two states does not need to be modified.

E) Multiple Different Repairs: Our repair process is able to find multiple physically possible suggestions, if they exist, as shown in Fig. 7(e). The task has the same liveness as Example 1 but there are no user-added safety constraints. The task can be repaired by modifying either the initial preconditions or final postconditions of the original skills.

F) Intermediate Liveness: Fig. 7(f) depicts a task with \( \varphi_s = \Box \phi (x_0 \land y_0) \land \Box \phi (x_1 \land y_1) \) where the repair suggestions involve modifying the intermediate states of a skill such that the task is satisfied during the execution of a skill, in contrast to prior work where task completion was only determined after skill execution. As shown in the fourth and fifth columns of Fig. 7(f), two possible suggested modifications are to modify one of the intermediate states of either skill such that it passes through \( \{ x_1, y_1 \} \). Modifying the final postcondition of one of the skills to end in \( \{ x_1, y_1 \} \) and the precondition of the other skill to start in \( \{ x_1, y_1 \} \) is another possible suggestion.

G) Nondeterministic Final Postcondition: We show an example involving a skill with nondeterministic final postconditions in Fig. 7(g) that does not necessarily reduce the nondeterminism, but can change it instead. The postcondition that ends in \( \{ x_0, y_0 \} \) can either be changed to \( \{ x_1, y_1 \} \) or \( \{ x_2, y_0 \} \). The modification to \( \{ x_1, y_1 \} \) does not remove nondeterminism, but simply changes it, while the change to \( \{ x_2, y_0 \} \) reduces nondeterminism.

H) User-Controlled Inputs: Our repair process can also handle specifications with user controlled propositions. Given a task \( \varphi_s = \Box \phi (\pi_{\text{react}} \rightarrow x_2 \land y_2) \land \Box \phi (\neg \pi_{\text{react}} \rightarrow x_0 \land y_0) \) and the skills in Fig. 7(a), we perform repair. The skill suggestions are the same, as in Example 1 and Fig. 7(a).

XI. PHYSICAL DEMONSTRATIONS

We demonstrate the generality of our approach by presenting two different physical robots (Baxter and Jackal), their abstractions, and tasks. We discuss what the physical implementation of modification entails for these systems and what the process is when moving to a new physical system.

A. Baxter Moving a Block

Here, the Baxter robot is moving a block on the table (see Fig. 8). The task is to repeatedly move the block from one side of the table to the other while avoiding the phone in the middle of the table.

Abstraction: We abstract the position of the green block using \( S_G = \{ \pi_{\text{left-bottom}}, \pi_{\text{left-top}}, \pi_{\text{right-bottom}}, \pi_{\text{right-top}}, \pi_{\text{front}}, \pi_{\text{back}}, \pi_{\text{middle}} \} \). Each proposition is grounded to a rectangular region of space such that \( \forall \pi_i, \pi_j, G(\pi_i) \cap G(\pi_j) = \emptyset \)
We determine the position of the block using an OptiTrack motion capture system. There are no user-controlled propositions (i.e., $I_s = \emptyset$).

**Skills**: The Baxter has two skills, L2R and R2L [see Fig. 8(a), and (b)]. The skill L2R goes from $\pi_{\text{left-bottom}}$ to $\pi_{\text{left-top}}$ to $\pi_{\text{middle}}$ to $\pi_{\text{right-top}}$ to $\pi_{\text{right-bottom}}$, and skill R2L goes from $\pi_{\text{right-bottom}}$ to $\pi_{\text{right-top}}$ to $\pi_{\text{front}}$ to $\pi_{\text{left-top}}$ to $\pi_{\text{left-bottom}}$. We created these skills by defining training trajectories, as shown in Fig. 8(a) and (b). For L2R, we sampled 100 waypoints from locations grounded to $\pi_{\text{left-bottom}}, \pi_{\text{left-top}}, \pi_{\text{right-top}}$, and $\pi_{\text{right-bottom}}$ to define the trajectories. The point grounded to $\pi_{\text{left-top}}$ is constrained to have the same $x$ and $y$ coordinates as the point grounded to $\pi_{\text{left-bottom}}$. The point grounded to $\pi_{\text{right-top}}$ is constrained to have the same $y$ and $z$ coordinates as the point grounded to $\pi_{\text{left-top}}$. The point grounded to $\pi_{\text{right-bottom}}$ is constrained to have the same $x$ and $y$ coordinates as the point grounded to $\pi_{\text{right-top}}$. Given these waypoints that define the trajectories, we train the weights for a DMP to reproduce the waypoints given a start and end point. When the robot executes a skill it finds the joint angles needed to reach each point in the trajectory (using Trak-IK inverse kinematics solver [42]) and then moves between the joint angles.

**Physical Feasibility**: To automatically validate physical feasibility, for the Baxter we consider two aspects: existence of an inverse kinematic solution, and no self-collisions. Given a trajectory (provided by the DMP that was modified to satisfy the symbolic repair) we use the Trak-IK inverse kinematics solver [42] to find the joint angles to achieve the points on the trajectory and then check that the proposed joint angles do not cause self-collision with the body or other arm.

**Task**: The task is to move the block between $\pi_{\text{right-bottom}}$ and $\pi_{\text{left-bottom}}$ while avoiding $\pi_{\text{middle}}$ (i.e., $\varphi^s = \Box \neg \pi_{\text{middle}} \land \Box \neg \pi_{\text{left-bottom}}$ and $\varphi^{\text{init}} = \Box \neg \pi_{\text{middle}} \land \Box \neg \pi_{\text{left-bottom}}$).

**Repair**: The task is not possible with these two skills. The symbolic repair first suggests a modification to L2R such that instead of passing through $\pi_{\text{middle}}$, it passes through $\pi_{\text{back}}$ and immediately goes to $\pi_{\text{right-bottom}}$. The robot is able to find a trajectory for the block that satisfies these new constraints [see Fig. 8(d)].

For this skill, our process determines that for the proposition $\pi_{\text{back}}$ to be $\text{True}$, the arm needs to move too close to the Baxter’s torso and thus there are self-collisions. We therefore automatically add $\Box \neg (\pi_{\text{left-top}} \land \pi_{\text{left-bottom}} \land \Box \neg \pi_{\text{back}})$ and $\Box \neg (\pi_{\text{back}} \land \pi_{\text{left-bottom}} \land \Box \neg \pi_{\text{right-bottom}})$ to $\varphi_{\text{disallowed-transitions}}$.

Continuing the repair, the new suggestion is to modify L2R such that it passes through $\pi_{\text{front}}$ instead of $\pi_{\text{middle}}$ and then goes directly to $\pi_{\text{right-bottom}}$. Our automated physical feasibility checking determines that the skill can be executed, as shown in Fig. 8(e). Fig. 8(f) shows the Baxter using this modified skill to accomplish the task.

**B. Jackal Patrolling and Reacting to Input From A Person**

We show a demonstration with a Jackal robot performing a patrolling task. Here we highlight two differences from the previous examples: the robot is reacting to input from a person, and the abstraction does not create a partition of the continuous state space, i.e., the grounding of some propositions overlap.

**Abstraction**: We abstract the position of the Jackal in the workspace using $I_G = \{\pi_0, \pi_B, \pi_C, \pi_D, \pi_E, \pi_F\}$. The propositions are grounded to either rectangles in $(X, Y)$ or circles. Furthermore, the propositions do not represent a partition of the workspace as in previous examples but rather show that the approach can be applied to overlapping groupings [see Fig. 9(b)]. Note that $\pi_B$ and $\pi_F$ are squares contained in the rectangle $\pi_B$. We tracked the position of the Jackal using a Vicon motion capture system. In addition, the user controls $I_s = \{\text{person}\}$ via a computer interface.

**Skills**: The robot has four skills. Each skill is generated by randomly sampling pairs of points from the precondition and postcondition and generating a curved trajectory between them. The weights of the DMP corresponding to the skill are learned from these initial trajectories. Fig. 9(a) shows the preconditions, postconditions, and original trajectories. For example, skill $0$ has precondition of $\pi_A$ and postcondition of $\pi_B$ (i.e., $\Sigma_{s, \text{pre-init}}$ skill $0 = \{\pi_A, \pi_B\}$, $\Sigma_{s, \text{post}, \text{final}}$ skill $0 = \{\pi_B\}$, $\pi_A, \pi_B, \pi_E, \pi_F$). After executing skill $0$, $\pi_B$ will always be $\text{True}$, but $\pi_A, \pi_E$, or $\pi_F$ may be $\text{True}$ as well, due to the overlap in the grounding sets. When a skill is executed, a point is randomly sampled from within the postconditions and passed to the DMP along with the current pose of the robot to generate the desired trajectory.

**Physical Feasibility**: To physically implement the skill, we control the Jackal to follow the waypoints of the trajectory using feedback linearization. Due to the small turning radius of the Jackal, we assume it can follow any sequence of waypoints.

**Task**: The task is $\varphi^s = \Box \neg (\pi_{\text{person}} \rightarrow \pi_E) \land \Box \neg (\neg \pi_{\text{person}} \rightarrow \pi_F) \land \Box \neg \pi_A \land \Box \neg \pi_D$; the robot should repeatedly visit $\pi_A$ and $\pi_D$.
\(\pi_D\), when the user input is True it should visit \(\pi_E\), and when the user input is False it should visit \(\pi_F\) [see Fig 9(c)]. This task does not contain safety constraints, and the requirements are of the form \(\square \Diamond \Box\), therefore, for the task to be achieved, the jackal needs to repeatedly, go through \(\pi_E\) when \(\pi_{\text{person}}\) is True and through \(\pi_F\) when it is False.

This task is unrealizable with the given skills because the robot cannot guarantee, due to the nondeterminism of the skills, that it will reach \(\pi_E\) or \(\pi_F\) to react to the user input.

**Repair:** The initial suggestion contains modifications to two skills, \(\text{skill} \rightarrow \pi_2 \land \pi_0 \rightarrow \pi_0\), and does not pass through \(\{\pi_B, \pi_F\}\). The repair process suggests modifying \(\text{skill} \rightarrow \pi_2\) such that it always passes through \(\{\pi_B, \pi_F\}\) and does not pass through \(\{\pi_B, \pi_E\}\). With the current parameters of weighing the constraints 50 times more than remaining close to the original trajectory and training for 100 epoch for the skill learning, we were unable to find a DMP that can generate trajectories that satisfy these constraints for more than 90% of the initial and final points. The second modification of the suggestion is for \(\text{skill} \rightarrow \pi_0 \rightarrow \pi_0\), which is physically possible. The repair process continues, incorporating the successful modification to \(\text{skill} \rightarrow \pi_0\), and returns another suggestion to modify \(\text{skill} \rightarrow \pi_0\), this time to result in \(\{\pi_B, \pi_E\}\) [see Fig. 9(e)]. This modification is also successful. With these two modified skills based on \(\text{skill} \rightarrow \pi_0\), we are able to successfully complete the task and react to input from a person.

**XII. DISCUSSION**

We have presented an automated approach to defining skills using GR(1) formulas and repairing high-level tasks in a physically feasible manner. Using GR(1) as the specification language allows us to specify tasks of general interest including conditional goals, and safety constraints.

Our algorithm is **Sound**; proposed symbolic suggestions will allow for successful task completion. All symbolic suggestions that the symbolic repair produces are guaranteed to allow the task to be completed; we verify the task can be done using synthesis. Our algorithm is also **symbolically complete**; if a symbolic repair exists given the constraints we have placed on acceptable suggestions, we will find it. The physical feasibility checking completeness depends on the algorithm we use. If all skills are correctly classified as physically possible or not following the attempted physical implementation, then the repair is complete. In the worst case, our repair algorithm will attempt every possible symbolic solution.

The computation time for the repair can be divided into the symbolic repair and the physical feasibility checking. The computation complexity of finding a symbolic repair is comparable to the complexity of the synthesis algorithm—polynomial in the state space. Practically speaking though, adding constraints on the possible skill modifications, ideally relating to the physical feasibility (e.g., do not teleport between areas) reduces the computation time. In our examples, symbolic repair was on the order of seconds to minutes. The physical skill modification time depends on the algorithm we choose; in this work it was on the order of minutes to train the DMP for the modified skill. Since this is an offline process, we used Python and did not optimize our code or the algorithms to be computationally efficient.

In this work we lay the foundation for physically feasible repair of high-level tasks. We leverage recent work on modifying DMPs to achieve LTL specifications [16]. However, our work is compatible with any skill learning/modification framework that would be amenable to modifications related to symbolic (LTL) constraints (e.g., [17], [45]); which we plan to explore in future work.

**XIII. CONCLUSION**

Within the context of creating complex robot behaviors through skill composition, in this work we address two challenges; taking into account intermediate states a skill may go through when physically executed by a robot, and providing an automated process for repairing infeasible tasks when there are missing skills.

We have shown that our approach generalizes to different robots, abstractions and tasks; but in order to do so, we need automated ways of creating controllers that satisfy LTL constraints. In this article, we use [16], in the future we will explore other approaches, such as the reinforcement learning methods as proposed in [28] or the modification of skills learned from demonstration proposed in [17]. Furthermore, given a robot, it is possible to automate the feasibility checking, for example using an IK solver, as we have shown with the Baxter. For other robots, we will explore the properties needed for such feasibility checking and how to suggest suitable techniques.

Another future direction is incorporating optimality criteria in the repair process. In this work we do allow the user to influence the symbolic repair by adding constraints on possible modifications. In the future, we will explore how continuous metrics can be specified and incorporated in the repair.

**APPENDIX**

The appendix provides formulas demonstrating the encoding of the skills and modifications in LTL formulas. In addition, we provide more detailed versions of Algorithms 1 and 2.

**Postconditions of L2R in \(\pi^{\text{mutable}}_e\) for Example 1 (2)**

\[
\begin{align*}
\square((\pi_{\text{L2R}} \land x_0 \land y_0 \land \neg x_1 \land \neg x_2 \land \neg y_1 \land \neg y_2) &\rightarrow \\
\circ (x_1 \land y_0 \land \neg x_0 \land \neg x_2 \land \neg y_1 \land \neg y_2)) &\land \\
\square((\pi_{\text{L2R}} \land x_1 \land y_0 \land \neg x_0 \land \neg x_2 \land \neg y_1 \land \neg y_2) &\rightarrow \\
\circ (x_2 \land y_0 \land \neg x_0 \land \neg x_1 \land \neg y_1 \land \neg y_2)) &\land \\
\square((\pi_{\text{L2R}} \land x_2 \land y_0 \land \neg x_0 \land \neg x_1 \land \neg y_1 \land \neg y_2) &\rightarrow \\
\circ (x_2 \land y_1 \land \neg x_0 \land \neg x_1 \land \neg y_0 \land \neg y_2)) &\land \\
\square((\pi_{\text{L2R}} \land x_2 \land y_1 \land \neg x_0 \land \neg x_1 \land \neg y_0 \land \neg y_2) &\rightarrow \\
\circ (x_2 \land y_2 \land \neg x_0 \land \neg x_1 \land \neg y_0 \land \neg y_1)) .
\end{align*}
\]
Algorithm 3: Modify_Postconditions.

Input: \( r \), \( r_{\text{hard}} \), \( r_{\text{mutable}} \), \( r_{\text{new}} \), \( Z \), \( L \)

Output: \( S_{r_{\text{new}}} \), \( S_{r_{\text{new}}-\text{mutable}} \)

// Find states where the system is guaranteed to reach \( Z \) for any environment choice
1. \( r_{\text{sys-can-win}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \times S_{\Sigma_{T}} \mid \exists s, s_{1}, s_{2}, s_{3}, s_{0}, 2 \in S_{O} \text{ s.t. } (s_{1}, 1, s_{0}, 1, s'_{2}) \in r_{\text{hard}} \text{ and } (s_{2}, s_{3}, 2) \in Z \}; \)
2. \( r_{\text{sys-always-wins}} = \{(s_{1}, 1, s_{0}) \in S_{\nu} \text{ s.t. } (s_{1}, 1, s_{0}, 2) \in r_{h} \rightarrow (s_{1}, 1, s_{0}, 2') \in r_{\text{sys-can-win}} \text{ and } r_{h} \neq \emptyset \}; \)
3. \( r_{\text{reachable}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid \exists s_{3}, s_{0}, 3 \in S_{O} \text{ s.t. } (s_{1}, 3, 3, 3, 3, 1) \in r_{\text{hard}} \text{ and } (s_{3}, s_{3}, 3, 3, 3) \in \tau_{h} \}; \)
4. \( r_{\text{always-wins and reachable}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in r_{\text{sys-always-wins}} \text{ and } (s_{1}, 1, s_{0}, 1) \in Z \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
5. \( r_{\text{allowed changes to preconditions}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in r_{\text{allowed changes to preconditions}} \}; \)
6. \( r_{\text{pre-skills}} = \{(s_{1}, 1, s_{0}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
7. \( r_{\text{final-post}} = \{(s_{1}, 1, s_{0}) \in S_{\nu} \mid \exists s_{3} \in S_{\Sigma_{T}} \text{ s.t. } (s_{1}, 3, 3, 3, 3, 1) \in \tau_{h} \}; \)
8. \( r_{\text{allowed changes}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in r_{\text{allowed changes}} \}; \)
9. \( r_{\text{selected-change} = \text{randomlySelect}}(r_{\text{allowed changes}}); \)
10. \( r_{\text{selected-change}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
11. \( r_{\text{allowed changes}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
12. \( r_{\text{selected-change}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
13. \( r_{\text{selected-change}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1) \in \tau_{h} \}; \)
14. \( r_{\text{return}} S_{r_{\text{new}}}, S_{r_{\text{mutable}}}; \)
15 end

// Add transitions to \( r_{\text{mutable}} \)
16. \( r_{\text{new-skills}} = \{(s_{3}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{3}, 1, s_{0}, 1, s'_{2}, 2) \in \tau_{h} \}; \)
17. \( r_{\text{old-skills}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in \tau_{h} \}; \)
18. \( r_{\text{selected-pre-conditions}} = \{(s_{1}, 1, s_{0}) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in \tau_{h} \}; \)
19. \( r_{\text{new-pre-conditions}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in \tau_{h} \}; \)
20. if \( (s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid \forall s_{1}, s_{0} \in S_{O} \text{ s.t. } (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \text{ and } (s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in \tau_{h} \) then
21. \( r_{\text{new}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
22. \( r_{\text{new-skills}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
23. \( r_{\text{new-pre-conditions}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
24. \( r_{\text{new}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
25 end
26. \( r_{\text{new}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
27. \( r_{\text{new-skills}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
28. \( r_{\text{new-pre-conditions}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
29. \( r_{\text{new}} = \{(s_{1}, 1, s_{0}, 1, s'_{2}, 2) \in S_{\nu} \mid (s_{1}, 1, s_{0}, 1, s'_{2}) \in \tau_{h} \}; \)
30. \( r_{\text{return}} S_{r_{\text{new}}}, S_{r_{\text{mutable}}}; \)
31 end
Algorithm 4: Modify Postconditions.

Input: \( \tau_e, \tau_{\text{new}}, \tau_{\text{mutable}}, \tau_{\text{hard}}, \tau_s, Z, \mathcal{T}_u \)

Output: \( \tau_{\text{new}}, \tau_{\text{mutable}} \)

// Find states where the system is guaranteed to reach Z for any environment choice
1. \( \tau_{\text{sys-can-win}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \Sigma_T \times \Sigma_O | \exists \sigma_{T,1}, \sigma_{Z,2}' \in \Sigma_T, \sigma_{O,1} \in \Sigma_O \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{mutable}} \text{ and } (\sigma_{Z,2}', \sigma_{O,2}') \in Z \} \)

2. \( \tau_{\text{sys-always-win}} = \{(\sigma_{T,1}, \sigma_{O,1}) \in \Sigma_T | \forall \sigma_{Z,2}' \in \Sigma_O \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \tau_e \rightarrow (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \tau_{\text{sys-can-win}}, \sigma_{O,1} \neq \emptyset \} \)

3. \( \tau_{\text{reachable}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \tau_e | \exists \sigma_{Z,3}, \sigma_{O,3} \in \Sigma_T, \sigma_{O,3} \in \Sigma_O \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{mutable}} \text{ and } (\sigma_{Z,3}, \sigma_{O,3}, \sigma_{Z,2}') \in Z \} \)

4. \( \tau_{\text{always-win-and-reachable}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}) \in \tau_{\text{sys-always-win}}, (\sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{reachable}}, (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{mutable}} \} \)

// Find allowed changes to preconditions
5. \( \tau_{\text{skills-changing}} = \{(\sigma_{O,1}) \in \Sigma_O | \exists \sigma_{T,1}, \sigma_{Z,2}' \in \Sigma_T, \sigma_{O,2} \in \Sigma_O \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{always-win-and-reachable}} \} \)

6. \( \tau_{\text{pre-post}} = \{(\sigma_{T,1}, \sigma_{O,1}) \in \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \tau_{\text{reachable}} \text{ and } \sigma_{O,1} \in \tau_{\text{skills-changing}} \} \)

7. \( \tau_{\text{final-post}} = \{(\sigma_{T,1}, \sigma_{O,1}) \in \Sigma_T \times \Sigma_O | \exists \sigma_{Z,3}, \sigma_{O,3} \in \Sigma_T \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{reachable}} \text{ and } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in Z \} \)

8. \( \tau_{\text{allowable-changes}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}', \sigma_{O,3}) \in \Sigma_T \times \Sigma_O \times \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{reachable}} \text{ and } \sigma_{O,1} \in \tau_{\text{skills-changing}} \} \)

9. \( \text{if } \tau_{\text{allowable-changes}} \neq \emptyset \text{ then} \)
10. \( \tau_{\text{selected-change}} = \text{randomlySelectOne}(\tau_{\text{allowable-changes}}) \)
11. \( \text{else } \)
12. \( \text{return } \tau_e, \tau_{\text{mutable}} \)

13. end

14. end

16. \( \tau_{\text{new-skill}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{selected-change}} \} \)

17. // Add transitions to \( \tau_{\text{mutable}} \)
18. \( \text{else } \)
19. \( \text{return } \tau_e, \tau_{\text{mutable}} \)

20. end

21. \( \tau_{\text{only-input-prop}} = \{(\sigma_{T,2}) \in \Sigma_T | (\sigma_{T,2}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{new-pre-condition}} \} \)

22. \( \tau_{\text{diff-skill-with-same-pre}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{mutable}} \text{ and } \sigma_{T,2} \in \tau_{\text{only-input-prop}} \} \)

23. \( \tau_{\text{remove}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \Sigma_T \times \Sigma_O | \exists \sigma_{O,2} \in \Sigma_O \text{ s.t. } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{mutable}} \text{ and } (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{diff-skill-with-same-pre and } \sigma_{T,2} \in \tau_{\text{only-input-prop}}} \} \)

24. \( \tau_{\text{new-pre-condition}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \Sigma_T \times \Sigma_O | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_{\text{remove}} \} \)

25. end

26. \( \tau_{\text{new}} = \tau_{\text{mutable}} \cup \tau_{\text{new-pre-condition}} \cup \tau_{\text{new-skill}} \)

27. // Add transitions to \( \tau_e \)
28. \( \tau_{\text{add-pre-new}} = \{(\sigma_{T,1}, \sigma_{O,1}) \in \Sigma_T | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \tau_{\text{reachable}}, (\sigma_{T,2}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{add-skill}}, (\sigma_{T,2}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,3}') \in \tau_{\text{new-skill}} \} \)

29. \( \text{else } \)
30. \( \tau_{\text{new-skill}} = \{(\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}') \in \Sigma_T | (\sigma_{T,1}, \sigma_{O,1}, \sigma_{Z,2}', \sigma_{O,2}') \in \tau_e \} \)

31. end
Preconditions of L2R in $\varphi_s^{mutable}$ for Example 1, and (3), (4)

$$
\varphi_s^{continue}
$$

Postconditions of a modified L2R in $\varphi_e^{mutable}$ for Example 1, as shown in Fig. 5 and (2)

$$
\varphi_s^{choose}
$$

Preconditions of a modified L2R in $\varphi_s^{mutable}$ for Example 1, as shown in Fig. 5, and (3), (4)

The symbolic suggestion for physical modifications to L2R in Example 1 shown in Fig. 5(d) and (6)

References

[1] G. Konidaris, L. P. Kaelbling, and T. Lozano-Perez, “From skills to symbols: Learning symbolic representations for abstract high-level planning,” J. Artif. Intell. Res., vol. 61, pp. 215–289, Jan. 2018.

[2] R. Tedrake, I. R. Manchester, M. Tobenkin, and J. W. Roberts, “LQR-trees: Feedback motion planning via sums-of-squares verification,” Int. J. Robot. Res., vol. 29, no. 8, pp. 1035–1052, Jul. 2010.

[3] A. Majumdar and R. Tedrake, “Funnel libraries for real-time robust feedback motion planning,” Int. J. Robot. Res., vol. 36, no. 8, pp. 947–982, Jul. 2017.

[4] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, “Sequential composition of dynamically dexterous robot behaviors,” Int. J. Robot. Res., vol. 18, no. 6, pp. 534–555, Jun. 1999.

[5] H. Kress-Gazit, M. Lahijanian, and V. Raman, “Synthesis for robots: Guarantees and feedback for robot behavior,” Annu. Rev. Control, Robot., Auton. Syst., vol. 1, pp. 211–236, May 2018.

[6] S. M. LaValle, Planning Algorithms. Cambridge, U.K.: Cambridge Univ. Press, 2006.

[7] R. E. Fikes and N. J. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” Artif. Intell., vol. 2, no. 3–4, pp. 189–208, Sep. 1971.

[8] A. Pacheco, S. Moarref, and H. Kress-Gazit, “Finding missing skills for high-level behaviors,” in Proc. IEEE Int. Conf. Robot. Automat., 2020, pp. 10335–10341.

[9] G. E. Finekos, “Revising temporal logic specifications for motion planning,” in Proc. IEEE Int. Conf. Robot. Automat., 2011, pp. 40–45.

[10] R. Alur, S. Moarref, and U. Topcu, “Counter-strategy-guided refinement of GR(1) temporal logic specifications,” in Proc. IEEE Formal Methods Comput.-Aided Des., 2013, pp. 26–33.

[11] C. J. Vasile, D. Aksary, and C. Belta, “Time window temporal logic,” Theor. Comput. Sci., vol. 691, pp. 27–54, 2017.

[12] S. Ghosh et al., “Diagnosis and repair for synthesis from signal temporal logic specifications,” in Proc. 19th Int. Conf. Hybrid Syst.: Comput. Control, 2016, pp. 31–40.

[13] A. T. Buyukkokoc and D. Aksary, “Temporal relaxation of signal temporal logic specifications for resilient control synthesis,” in Proc. IEEE 61st Conf. Decis., 2022, pp. 2890–2896.
[14] D. Kamale, E. Karyofylli, and C.-I. Vasile, “Automata-based optimal planning with relaxed specifications,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., 2021, pp. 6525–6530.

[15] V. Raman and H. Kress-Gazit, “Explaining impossible high-level robot behaviors,” IEEE Trans. Robot., vol. 29, no. 1, pp. 94–104, Feb. 2013.

[16] C. Innes and S. Ramamoorthy, “Elaborating on learned demonstrations with temporal logic specifications,” in Proc. Robot.: Sci. Syst., Corvalis, OR, USA, 2020.

[17] M. A. Rana, M. Mukadam, S. R. Ahmadzadeh, S. Chernova, and B. Boots, “Towards robust skill generalization: Unifying learning from demonstration and motion planning,” in Proc. 2017 Conf. Robot Learn., 2017, pp. 109–118.

[18] R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Teaching multiple tasks to an RL agent using LILT,” in Proc. 17th Int. Conf. Auton. Agents MultiAgent Syst., 2018, pp. 452–461.

[19] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa’ar, “Synthesis of reactive (1) designs,” J. Comput. System Sci., vol. 78, no. 3, pp. 911–938, May 2012.

[20] R. Königshofer, G. Hofferek, and R. Bloem, “Debugging formal speciﬁcations using simple counterstrategies,” in Formal Methods in Comput.-Aided Des., 2009, pp. 152–159.

[21] K. Kim, G. Fainekos, and S. Sankaranarayanan, “On the minimal revision problem of speciﬁcation automata,” Int. J. Robot. Res., vol. 34, no. 12, pp. 1515–1535, Oct. 2015.

[22] A. Dokhanchi, B. Hoxha, and G. Fainekos, “Metric interval temporal logic speciﬁcation elicit and debugging,” in Proc. ACM/IEEE Int. Conf. Formal Methods Models Codesign, 2015, pp. 70–79.

[23] K. Chatterjee, B. Henzinger, and A. T. Jobstmann, “Environment assumptions for synthesis,” in Proc. 19th Int. Conf. CONCUR-Concurrency Theory, Berlin, Heidelberg, 2008, pp. 147–161.

[24] W. Li, L. Dworkin, and S. A. Seshia, “Mining assumptions for synthesis,” in Proc. IEEE/ACM Int. Conf. Formal Methods Models Codesign, 2011, pp. 43–50.

[25] M. Lahijanian, S. B. Andersson, and C. Belta, “Temporal logic motion planning and control with probabilistic satisfaction guarantees,” IEEE Trans. Robot., vol. 28, no. 2, pp. 396–409, Apr. 2012.

[26] M. Lahijanian, M. R. Malay, D. Fried, L. E. Kavraki, H. Kress-Gazit, and M. Y. Vardi, “Iterative temporal planning in uncertain environments with partial satisfaction guarantees,” IEEE Trans. Robot., vol. 32, no. 3, pp. 583–599, Jun. 2016.

[27] A. Pacheck, G. Konidaris, and H. Kress-Gazit, “Automatic encoding and repair of reactive high-level tasks with learned abstract representations,” in Proc. 19th Int. Symp. Robot. Res., 2019, pp. 509–525.

[28] X. Li, C.-I. Vasile, and C. Belta, “Reinforcement learning with temporal logic rewards,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., 2017, pp. 3834–3839.

[29] L. Illanes, X. Yan, R. T. Icarte, and S. A. McIlraith, “Symbolic plans as high-level instructions for reinforcement learning,” in Proc. Int. Conf. Automated Plan. Scheduling, 2020, vol. 30, pp. 540–550.

[30] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith, “LILT and beyond: Formal languages for reward function speciﬁcation in reinforcement learning,” in Proc. Int. Joint Conf. Artif. Intell., 2019, vol. 19, pp. 6065–6073.

[31] C. R. Garrett et al., “Integrated task and motion planning,” Annu. Rev. Control, Robot., Auton. Syst., vol. 4, no. 1, pp. 265–293, May 2021, doi: 10.1146/annurev-control-091420-084139.

[32] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, “Combined task and motion planning through an extensible planner-independent interface layer,” in Proc. IEEE Int. Conf. Robot. Automat., 2014, pp. 639–646.

[33] M. Mahdavyuddin, L. M. Kavraki, and J. Rossel, “Randomized physics-based motion planning for grasping in cluttered and uncertain environments,” IEEE Robot. Automat. Lett., vol. 3, no. 2, pp. 712–719, Apr. 2018.

[34] J. Lee, C. Nam, J. Park, and C. Kim, “Tree search-based task and motion planning with prehensile and non-prehensile manipulation for obstacle rearrangement in clutter,” in Proc. IEEE Int. Conf. Robot. Automat., 2021, pp. 8516–8522.

[35] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, “An incremental constraint-based framework for task and motion planning,” Int. J. Robot. Res., vol. 37, no. 10, pp. 1134–1151, Sep. 2018, doi: 10.1177/0278364918761570.

[36] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard, “Recent advances in robot learning from demonstration,” Annu. Rev. Control, Robot., Auton. Syst., vol. 3, pp. 297–330, 2020.

[37] A. Pnueli, “The temporal logic of programs,” in Proc. IEEE 18th Ann. Symp. Foundations Comput. Sci., 1977, pp. 46–57.

[38] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert, “Learning motion primitives,” in Proc. Robot. Res. 11th Int. Symp., Berlin, Heidelberg, 2005, pp. 561–572.

[39] N. Jaquier, D. Ginsbourger, and S. Calinon, “Learning from demonstration with model-based Gaussian process,” in Proc. Conf. Robot Learn., 2019, pp. 247–257.

[40] M. Noseworthy, R. Paul, S. Roy, D. Park, and N. Roy, “Task-conditioned variational autoencoders for learning motion primitives,” in Proc. Conf. Robot Learn., 2019, pp. 933–944.

[41] T. Kulak, J. Silverio, and S. Calinon, “Fourier motion primitives: An approach for learning rhythmic robot skills from demonstrations,” in Proc. Robot.: Sci. Syst., Corvalis, Oregon, USA, 2020.

[42] P. Beeson and B. Ames, “TRAC-IK: An open-source library for improved solving of generic inverse kinematics,” in Proc. IEEE-RAS 15th Int. Conf. Humanoid Robots, 2015, pp. 928–935.

[43] O. Maler and D. Nickovic, “Monitoring temporal properties of continuous signals,” in Proc. Formal Techn., Modelling Anal. Timed Fault-Tolerant Syst.: Joint Int. Conf. Formal Model. Anal. Timed Syst., FORMATS, Formal Techn. Real-Time Fault-Tolerant Syst., 2004, pp. 152–166.

[44] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability,” Formal aspects Comput., vol. 6, pp. 512–535, 1994.

[45] A. Paraschos, C. Daniel, J. R. Peters, and G. Neumann, “Probabilistic movement primitives,” Adv. Neural Inf. Process. Syst., vol. 26, pp. 2616–2624, 2013.