Warm-Starting Fixed-Point Based Control Synthesis

Zexiang Liu Necmiye Ozay
EECS
University of Michigan
Ann Arbor, United States
zexiang@umich.edu necmiye@umich.edu

Nicolas Perrin
CNRS UMR 7222, ISIR
Sorbonne Université
Paris, France
perrin@isir.upmc.fr

In this work we propose a patching algorithm to incrementally modify controllers, synthesized to satisfy a temporal logic formula, when some of the control actions become unavailable. The main idea of the proposed algorithm is to "warm-start" the synthesis process with an existing fixed-point based controller that has a larger action set. By exploiting the structure of the fixed-point based controllers, our algorithm avoids repeated computations while synthesizing a controller with restricted action set. Moreover, we show that the algorithm is sound and complete, that is, it provides the same guarantees as synthesizing a controller from scratch with the new action set. An example on synthesizing controllers for a simplified walking robot model under ground constraints is used to illustrate the approach. In this application, the ground constraints determine the action set and they might not be known a priori. Therefore it is of interest to quickly modify a controller synthesized for an unconstrained surface, when new constraints are encountered. Our simulations indicate that the proposed approach provides at least 5-times speed-up compared to synthesizing a controller from scratch.

1 INTRODUCTION

Control synthesis techniques for discrete systems have been a central topic both for reactive synthesis and discrete-event systems [13, 14] with recent results establishing a connection between the two communities [4, 15]. These techniques provide a principled means for computing a controller with correctness guarantees for systems that can either be directly modeled as or whose continuous-dynamics can be abstracted as a discrete transition system. Such discrete controllers are ubiquitous in many embedded applications.

A key challenge in control synthesis is scalability. The scalability depends both on the size of the discrete transition system and the complexity of the specification, e.g., can be doubly exponential in the length of a general linear temporal logic (LTL) specification [13]. Therefore, some research has focused on identifying fragments of LTL that have favorable complexity (see e.g., [2, 3, 17, 12]). Although these fragments lead to tractable problems, the time required for synthesis still prevents it from being applicable on-line, necessitating to consider all possible scenarios at design-time. This motivates the following questions: (i) If controllers are to be synthesized off-line for many different scenarios, and if there is a controller for a specific scenario in hand, can this controller be used to synthesize new controllers efficiently? (ii) can this modification approach be fast enough to enable on-line synthesis if new situations are encountered at run-time?

The problem of incrementally modifying an existing controller, i.e., "patching" a controller, as opposed to re-synthesizing from scratch, has been studied for different control synthesis techniques, specifically in the context of robotics applications [10, 9, 18]. Livingston et al. [10] study a patching method for two player games with the specifications given by the GR(1) fragment of LTL and the control strategies synthesized by a μ-calculus based method. Assuming that changes in the system and environment only break an existing controller locally, they re-synthesize a controller only for the affected nodes and

Presented at:
SYNT 2018

replace the broken part with the new one. This method is also extended to handle changes in the specification such as addition of new goal regions [9]. Wong et al. [18] also consider GR(1) specifications with corresponding symbolic controller synthesis techniques and develop recovery mechanisms when the environment assumptions become invalid during execution.

This paper also considers the problem of patching controllers. Different from the existing literature mentioned above, we work with synthesis problems where there exists an explicit transition system to be controlled and the modification required is due to some of the control actions becoming unavailable. The loss of control actions is relevant in the case of actuator faults and also in the context of stabilizing a walking robot on a constrained surface, as described later, or more generally of systems with action constraints that vary with time or environment changes. Another difference is the class of specifications considered: the LTL fragment we use includes persistence requirements, which are not expressible within the GR(1) fragment, and is amenable to fixed-point based control synthesis techniques operating directly on the transition system (see e.g., [17, 12]). Our main contribution is to propose a novel *controller implementation*, a data structure consisting of a partially ordered set of controllers corresponding to simple fixed-points in the synthesis algorithm, that captures all the information required for modification when some of the actions become unavailable. We then present patching techniques using this data structure that modify it appropriately to compute the new controller. The proposed patching techniques are in a sense similar to warm-starting techniques in optimization, where an existing solution (not too far away from the expected new solution) is used to initialize an optimization algorithm. Similarly, we use an existing controller to initialize the synthesis algorithm for finding a controller for the new problem.

We demonstrate the proposed approach on a push recovery example for a 1D walking robot model. The potential control actions for the robot are the feasible foot placements. However, if there are ground constraints, e.g., holes, obstacles, on which the robot is not allowed to step, the available action set reduces. Such constraints might not be available a priori, hence, it is of interest to design many controllers for different sets of ground constraints. In addition, if one wants to use 1D push recovery controllers to navigate a 2D surface with constraints, it is again possible (albeit with some conservatism) to consider a large number of 1D ground constraint profiles and corresponding family of 1D controllers for the 2D navigation task. Our results for this example show that up to an order of magnitude speed-up can be achieved if the controllers for constrained surface are generated by patching the controller for the unconstrained surface.

The paper is organized as follows: In Section 2, we give the problem statement and the background knowledge about fixed-point based control synthesis. Section 3 presents the main algorithm that warm-starts a control synthesis from an existing controller, and the proof of soundness and completeness. We use a simple running example in Sections 2 and 4 to illustrate how the method works before presenting an example on control synthesis for a walking robot in Section 4 that shows the time efficiency and potential applicability of our method.

2 PRELIMINARIES

2.1 Notation

For two sets A and B, the set difference is denoted by $A - B$. Given $T \subseteq A \times B \times C$, for a set $E \subseteq A$, $(E,*,*)_T$ refers to the set $(E \times B \times C) \cap T$, and similarly for $F \subseteq B$ and $G \subseteq A \times B$, $(*,F,*)_T$ and $(G,*)_T$ refer to the sets $(A \times F \times C) \cap T$ and $(G \times C) \cap T$ respectively. A list $\mathscr{V} = [x_1,...,x_n]$ is a totally ordered set, where $\mathscr{V}(i)$ refers to its ith-order element. To assign x_i to be the ith element in $\mathscr{V}$, we write $\mathscr{V}(i) = x_i$.

2.2 Augmented Finite Transition Systems

We consider plants modeled as augmented finite transition systems, a discrete structure that either can be used as a direct modeling tool or can be obtained by abstracting the dynamics of a continuous-control system [12].

Definition 1. An **augmented finite transition system** (AFTS) is a tuple $T = (Q, U, \rightarrow_T, G, AP, h_Q)$, where Q is a set of states, U is a set of actions (control inputs), $\rightarrow_T \in Q \times U \times Q$ is a transition relation between states under specific actions, $G : 2^U \rightarrow 2^{2^Q}$ maps a set of actions to a set of progress groups[1]under that action set, AP is a finite set of atomic propositions, and $h_Q : Q \rightarrow 2^{AP}$ is a labeling function, which maps each state to the set of atomic propositions that evaluate to true at that state.

Without loss of generality for a given AFTS, we assume that all the actions are available at each state.

A *trajectory* of an AFTS T is an infinite sequence of states $\{s(n)\}_{n=0}^{\infty}$ such that for any two consecutive states $s(n), s(n+1)$ in the sequence, there exists an action $a \in U$ satisfying $(s(n), a, s(n+1)) \in \rightarrow_T$ and the sequence is compliant with the progress groups.

2.3 Linear Temporal Logic

Linear Temporal Logic (LTL) is utilized to describe the desired behaviors of an AFTS. It consists of logic operators (negation $\neg$, conjunction $\wedge$, disjunction $\vee$), and temporal operators (next $\bigcirc$, always $\square$, eventually $\diamond$ and until $\mathbf{U}$).

Syntax of LTL [1]: The LTL formula over a finite set of atomic propositions AP can be formed according to the grammar:

$$\phi := True \mid p \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \bigcirc \phi \mid \phi_1 \mathbf{U} \phi_2$$

where $p \in AP$, ϕ_1 and ϕ_2 are also LTL formulas. The other operators can be derived as follows: $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$, $\phi_1 \implies \phi_2 = \neg\phi_1 \vee \phi_2$, $\diamond\phi = True\ \mathbf{U}\ \phi$, $\square\phi = \neg\diamond\neg\phi$.

Semantics of LTL [12]: An ω-word is an infinite sequence in 2^{AP}. The satisfaction of an LTL specification ϕ at position i by an ω-word $w = w(0)w(1)w(2)\ldots$, written as $(w,i) \models \phi$, is defined inductively as follows:

- For $\phi = p \in AP$, $(w,i) \models p$ iff $p \in w(i)$
- $(w,i) \models \neg\phi$ iff $(w,i) \not\models \phi$
- $(w,i) \models \phi_1 \vee \phi_2$ iff $(w,i) \models \phi_1$ or $(w,i) \models \phi_2$
- $(w,i) \models \bigcirc\phi$ iff $(w,i+1) \models \phi$
- $(w,i) \models \phi_1 \mathbf{U} \phi_2$ iff $\exists j \geq i$ s.t. $(w,j) \models \phi_2$ and $(w,k) \models \phi_1, \forall k \in [i,j)$

An ω-word w satisfies ϕ if and only if $(w,0) \models \phi$, written as $w \models \phi$.

Given a trajectory $\{s(n)\}_{n=0}^{\infty}$ of an AFTS, the *ω-word* corresponding to $\{s(n)\}_{n=0}^{\infty}$ is $\{h_Q(s(n))\}_{n=0}^{\infty}$, where h_Q is the labeling function of the AFTS. Given a LTL specification ϕ, we say that $\{s(n)\}_{n=0}^{\infty} \models \phi$ if and only if $\{h_Q(s(n))\}_{n=0}^{\infty} \models \phi$.

[1] A set $G \in G(D)$ is called a *progress group under the action set D* , and it is related to the following semantic notion: the system cannot remain indefinitely within G by exclusively choosing actions from U. This restricts the behaviors allowed by the transition relation. Moreover, it is assumed that G is not invariant under the actions U for the well-formedness of the definition (see [12, 16, 8]).

2.4 Control Synthesis and Fixed-Point Operators

The LTL specification considered in this work is of the form:

$$\phi = \Box A \wedge \Diamond\Box B \wedge \left(\bigwedge_{i \in I} \Box\Diamond R^i \right) \tag{1}$$

for atomic propositions A, B, R^i. Such specifications can express properties of *invariance* ($\Box A$), *persistence* ($\Diamond\Box B$) and *recurrence* ($\Box\Diamond R^i$) . This LTL fragment, also used in [17, 12], is a subset of Generalized Rabin specifications [4]. As the progress groups in AFTS is equivalent to environment liveness assumptions, the formula in (1) can also be seen as a generalization of GR(1) specifications, with well-separated environments [7, 11, 15] under the well-formedness assumption in footnote 1. In what follows, with a slight abuse of notation, we treat A, B, R^i as subsets of the state set Q, e.g., $A = \{s \in Q \mid A \in h_Q(s)\}$, etc.

Roughly speaking, the control synthesis problem for an AFTS is to compute a function, i.e., control strategy, that restricts the possible actions each time a state is visited so that only the desirable trajectories of the AFTS remain. A control strategy is formally defined next.

Definition 2. A **control strategy** for an AFTS is a partial function $\mu : (Q \times U)^* \times Q \to 2^U$ that maps the history of state-action pairs and the current state to a set of actions.

For a given LTL specification, one can talk about the "best" control strategy in enforcing the specification and the set of initial states for which such a strategy is defined, i.e. the winning set.

Definition 3. The **winning set** for a specification ϕ over an AFTS $T = \{Q, U, \to_T, G, AP, h_Q\}$, written as W_ϕ, is the largest subset of Q such that if the system T is initially in W_ϕ, the specification can be enforced.

Note that a control strategy as in definition 2 can require infinite memory. However for LTL specifications, it is known that there exists a finite memory control strategy corresponding to the winning set. In particular, for the specifications of the form (1), we define a data structure, which we call *controller*, to implement a control strategy in a specific way. This data structure will be crucial for the patching algorithms developed in Section 3.

Definition 4. A **simple controller** for a set $D \subseteq Q$ over an AFTS $T = \{Q, U, \to_T, G, AP, h_Q\}$ is a function $\mathscr{C} : D \to 2^U$, i.e. a memoryless controller that maps states in D to a set of actions.

Given $\widehat{D} \subseteq D$, a simple controller $\mathscr{C}$ **restricted to** $\widehat{D}$ means that the domain of $\mathscr{C}$ is restricted to $\widehat{D}$.

Definition 5. For the AFTS $T = \{Q, U, \to_T, G, AP, h_Q\}$, a **controller** $\mathscr{C}$ is a tuple $(\mathscr{V}, \mathscr{K}, x)$, where $\mathscr{V}$ is a list of subsets of Q, $\mathscr{K}$ is a list of controllers or simple controllers, which we refer to as sub-controllers of $\mathscr{C}$, and x is an internal variable that indicates the index of sub-controllers executed last time.

Overall, a controller is a tree structure with controllers at each node and simple controllers at the leaf nodes. The list $\mathscr{K}$ for each controller (each non-leaf node in the tree) denotes the children of that controller in the tree. Winning sets and controllers for specifications in the form of (1) are computed iteratively via fixed-point based algorithm (9) given in Appendix A.

Definition 6. Execution of controllers: In the execution time, a controller $\mathscr{C} = (\mathscr{V}, \mathscr{K}, x)$ acts as a function that maps the current state in Q to a set of feasible actions in U, for the AFTS $T = (Q, U, \to_T, G, AP, h_Q)$. Initially the internal variables are set to 1. Given the current state s, the output $\mathscr{C}(s)$ is determined in two steps:

(i) update the internal variable x of $\mathscr{C}$:

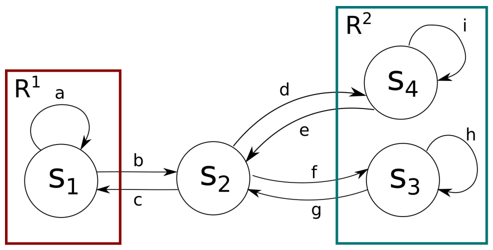


Figure 1: A simple AFTS with state space $\{s1, s2, s3, s4\}$ and action space $\{a, b, c, ..., h\}$, used to illustrate how the controller works. Atomic propositions are $R^1 = \{s_1\}$ and $R^2 = \{s_3, s_4\}$. For simplicity, assume no progress group exists.

– if $\mathscr{C}$ results from (9), (11), (14) (see Appendix A):

$$x^+ = \begin{cases} \arg\min_{y \le x}\{s \in \mathscr{V}(y)\}, & \text{if } x \neq 1 \\ \arg\min_{y}\{s \in \mathscr{V}(y)\}, & \text{otherwise} \end{cases}$$

– if $\mathscr{C}$ results from (10) (see Appendix A):

$$x^+ = \begin{cases} (x \mod |\mathscr{K}|) + 1, & \text{if } s \in \mathscr{V}(x+1) \\ x, \text{ otherwise} \end{cases}$$

where x^+ refers to the value of internal variable x after update.

(ii) Execute the x^+th sub-controller and return its output:

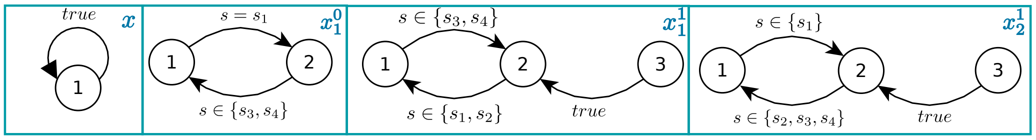
$$\mathscr{C}(s) = \mathscr{K}(x^+)(s)$$

In the execution of $\mathscr{K}(x^+)(s)$, the process above is repeated and then a sub-controller of $\mathscr{K}(x^+)$ is executed. Thus it is a recursive process that does not end until a simple controller is reached. The set of feasible actions are returned by that simple controller. For the sub-controllers not called in this execution, their internal variables remain unchanged.

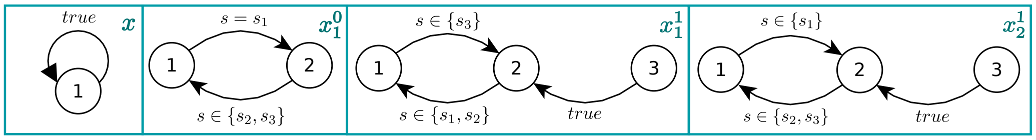
Example 1. For a simple AFTS shown in Figure 1, given the specification $\phi = \Box\Diamond R^1 \wedge \Box\Diamond R^2$, the outputs of (9) are the winning set $W = \{s_1, s_2, s_3, s_4\}$, and the controller $\mathscr{C} = (W, \{\mathscr{C}_1^0\}, x = 1)$, where its descendant controllers[2] are:
$\mathscr{C}_1^0 = (\{W, \{s_1\}, \{s_3, s_4\}\}, \{\mathscr{C}_1^1, \mathscr{C}_2^1\}, x_1^0)$; $\mathscr{C}_1^1 = (\{\{s_1, s_2\}, W, W\}, \{\mathscr{C}_i^{2,0}\}_{i=1}^3, x_1^1)$;
$\mathscr{C}_2^1 = (\{\{s_2, s_3, s_4\}, W, W\}, \{\mathscr{C}_i^{2,1}\}_{i=1}^3, x_2^1)$; $\mathscr{C}_1^{2,0} = \{(s_1, \{a\}), (s_2, \{c\})\}$;
$\mathscr{C}_2^{2,0} = \{(s_1, \{a, b\}), (s_2, \{c\}), (s_3, \{g\}), (s_4, \{e\})\}$; $\mathscr{C}_3^{2,0} = \{(s_1, \{a, b\}), (s_2, \{c, d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$;
$\mathscr{C}_1^{2,1} = \{(s_2, \{d, f\}), (s_3, \{h\}), (s_4, i)\}$; $\mathscr{C}_2^{2,1} = \{(s_1, \{b\}), (s_2, \{d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$;
$\mathscr{C}_3^{2,1} = \{(s_1, \{a, b\}), (s_2, \{c, d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$.

$\mathscr{C}$, $\mathscr{C}_1^0$ and $\mathscr{C}_i^1$ result from (9), (10) and (11). $\mathscr{C}_i^{2,k}$ are simple controllers resulting from (13). Each time $\mathscr{C}$ is called, (x, x_1^0, x_1^1, x_2^1) is updated according to Definition 6. The simple controllers in $\mathscr{C}$ reached at that time determines the set of feasible control inputs. The update rules for internal variables in this example 1 can be illustrated by the transition graph in Figure 2, which is consistent with Definition 6.

[2]In this example, in Section 4 and Appendix A, a simple controller $\mathscr{C}_{sim}$ is denoted by the set $\{(s, \mathscr{C}_{sim}(s)) : s \in D\}$ for simplicity, where D is the domain of $\mathscr{C}_{sim}$.



(a) no action disabled



(b) action e is disabled

Figure 2: The transition graph here is a visual illustration of the update rules of x in Definition 6. (a) corresponds to the example 1. (b) corresponds to the example in Section 4.1. s is the current state of the system in Figure 1. At each execution time, each internal variable transits from its current node to one of the neighbor node once the transition condition on the edge is satisfied. If no transition condition is satisfied, internal variable remains unchanged. Update x first, and then x^0, and lastly x^1_0 and x^1_1.

For example, initialize the internal variables to be $(1,1,1,1)$. Let's start from s_1 and run $\mathscr{C}(s_1)$: The updated internal variables are $(1,2,1,2)$, after $\mathscr{C}(s_1), \mathscr{C}^0_1(s_1)$, $\mathscr{C}^1_2(s_1)$ and $\mathscr{C}^{2,1}_2(s_1)$ are called recursively. The set of feasible actions is $\{b\} = \mathscr{C}^{2,1}_2(s_1)$. Under action b, the system in Figure 1 transits to s_2. Now run $\mathscr{C}(s_2)$. The internal variables are updated to $(1,2,1,1)$. The set of feasible actions is $\{d,f\} = \mathscr{C}^{2,1}_1(s_2)$. Let's keep doing this and always take the first action in the output of $\mathscr{C}$ at each execution. The trajectory under control of $\mathscr{C}$ would be $s_1, s_2, s_4, s_2, s_1, \ldots$, where the sequence of actions is $b, d, e, c, \ldots$. The trajectory visits A and B in turn, satisfying the given specification.

2.5 Problem Statement

In this section, we formally define the problem of interest.

Problem 1. Given an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$, an LTL specification $\phi = \Box A \wedge \Diamond\Box B \wedge \left(\bigvee_{i\in I} \Box\Diamond R^i\right)$, the winning set W_ϕ and controller $\mathscr{C}_\phi$, if a set of actions $U_d \subseteq U$ in T is unavailable, find the new winning set and a controller with an algorithm that exploits the knowledge of W_ϕ and $\mathscr{C}_\phi$.

We start by giving an overview of our solution approach. Recall that a controller in Definition 5 has a tree structure whose nodes consist of controllers (non-leaf nodes) and simple controllers (leaf nodes): the non-leaf nodes are computed by algorithm (10), (11) and (12); the leaf nodes are computed by algorithm (13) and (14) (see Appendix A). Except for (9), each algorithm in the appendix corresponds to one underlying specification, specified by its input parameters. In Section 3, we show that each algorithm in the appendix has a corresponding patching algorithm (denoted with an over-bar on the operator for the original algorithm) which restricts an existing controller resulting from it to a smaller action space for a stronger specification [3]. These patching algorithms allow us to offer a solution to Problem 1 by patching the descendant controllers of a controller resulting from algorithm (9) one by one. Furthermore, since there is an order between the nodes in each layer of the tree structure (recall that $\mathscr{K}$ is a list), they need to be patched in the order that they are generated.

[3] In general, for specifications ϕ and ψ, "ϕ is stronger than ψ" means that whenever ϕ is satisfied, ψ is satisfied. Here a stronger specification always refers to the case that the target set Z in the specification of a fixed-point based algorithm shrinks (if its specification has such a term, see Section 3).

To get the intuition, consider a simple invariance specification ($\Box A$) and the corresponding winning set $W_{\Box A}$ computed through a simple contracting fixed-point that starts with the set A and shrinks this set until it finds the maximal invariant set that is contained in A. In this case the controller $\mathscr{C}_{\Box A}$ is just a chain (instead of an arbitrary tree) with a single simple controller. If the action set is reduced, then one can use the same simple contracting fixed-point with the new action set this time initialized with the set $W_{\Box A}$. Since $W_{\Box A}$ in general is smaller than A, the new fixed point algorithm needs less iterations. Our patching algorithm generalizes this idea to nested fixed-points where we systematically restrict the inputs to each of the algorithms recursively, using the controller data structure introduced.

3 PATCHING METHOD

Before presenting the patching methods, we state several properties of winning sets, which are vital to the success of patching existing controllers.

Given an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$ and an LTL specification ϕ, compute its winning set W_ϕ and the winning control strategy $\mathscr{C}_\phi$ via the corresponding algorithm listed in Appendix A. Now disable the set of actions $U_d \subseteq U$ in the AFTS T, which results in a new AFTS $\widehat{T} = (Q, \widehat{U}, \rightarrow_{\widehat{T}}, G, AP, h_Q)$, where $\widehat{U} = U - U_d$ and $\rightarrow_{\widehat{T}} = \rightarrow_T - \{(s_1, a, s_2) : s_1, s_2 \in Q, a \in U_d, (s_1, a, s_2) \in \rightarrow_T\}$. Re-compute the winning set and controller , denoted by $\widehat{W}_\phi$ and $\widehat{\mathscr{C}_\phi}$, for a specification the same as or stronger than ϕ over $\widehat{T}$ via the same algorithm. Here a stronger specification always refers that input parameter Z of algorithms (10), (11), (12) and (14) shrinks to a smaller set $\widehat{Z} \subseteq Z$ (the other parameters are kept the same) after U_d is removed from the action space. Typically Z is the target set we want to reach, which shrinks due to the unavailable actions. Then, we have the following theorems, proofs of which are omitted for brevity:

Theorem 1. For algorithm (14), we have $(\widehat{W}_\phi \cup \widehat{Z}) \subseteq (W_\phi \cup Z)$.

Theorem 2. For algorithms (9), (10), (11), (12) and (13), we have $\widehat{W}_\phi \subseteq W_\phi$.

If we fix the specification to be in the form of (1), Theorems 1 and 2 say that the winning sets of the controller and its nodes after U_d is disabled are bounded by winning sets of the controller and its nodes before U_d is disabled (or by the union of winning sets and Z for simpler controllers resulting from (14)), which is the basis for our warm-starting synthesis method. Based on this fact, it is possible to search the feasible control strategies inside the existing controllers, i.e. patching the existing controllers, instead of re-synthesizing from scratch, for they contain the control strategies for a larger winning set than we need. Actually, we show that all the feasible control strategies for the new problem setting can be obtained by patching the existing controllers in the next two sections.

3.1 Patching Simple Controllers

A controller is a tree structure built on simple controllers, i.e. the leaf nodes in the tree structure. Thus we need to patch the leaf nodes firstly, and then patch their parent nodes, and so forth. In this section, we discuss the patching algorithms for the simple controllers, and then in the following section we will show how to patch the other nodes building on the leaf nodes, and the controller resulting from (9).

Definition 7. Given a simple controller $\mathscr{C}$ over an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$, a **finite transition system** (FTS) corresponding to $\mathscr{C}$ is $A(\mathscr{C}) = (Q_C, U_C, \rightarrow_{A(\mathscr{C})})$, where Q_C is the set of states that appear in $\rightarrow_{A(\mathscr{C})}$, $U_C = \{u : u \in \mathscr{C}(s), \forall s \in D\}$, $\rightarrow_{A(\mathscr{C})} = \{(s_1, u, s_2) \in \rightarrow_T : s_1 \in D, s_2 \in Q, u \in \mathscr{C}(s_1)\}$, and D is the domain of $\mathscr{C}$.

Table 1: Basic Operations on the FTS corresponding to controller $\mathscr{C}$

Operation	Definition	Notation
transition removing	given $E \subseteq Q \times U \times Q$, replace $\rightarrow_{A(\mathscr{C})}$ with $\rightarrow_{A(\mathscr{C})} - E$	$A(\mathscr{C}) \backslash E$
transition adding	given $E \subseteq Q \times U \times Q$, replace $\rightarrow_{A(\mathscr{C})}$ with $\rightarrow_{A(\mathscr{C})} \cup E$	$A(\mathscr{C}) \cup E$
transition pre	given $S_2 \subseteq Q$, output $\{(s_1,a) : \exists s_2 \in S_2, (s_1,a,s_2) \in \rightarrow_{A(\mathscr{C})}\}$	$\widehat{\text{Pre}}^{A(\mathscr{C})}_{\exists,\exists}(S_2)$

Remark 1. By Definition 7, $A(\mathscr{C})$ can be easily constructed from the simple controller $\mathscr{C}$ and the AFTS T. Conversely, $\mathscr{C}$ can be constructed from $A(\mathscr{C})$ by $\mathscr{C} = \{(s, D(s)) : s \in W, D(s) \neq \emptyset\}$, where $D(s) = \{u : (s,u,s_2) \in \rightarrow_{A(\mathscr{C})}\}$.

$A(\mathscr{C})$ contains the transitions behind the state-action pairs in $\mathscr{C}$, so that we can search invalid transitions in $A(\mathscr{C})$ after some actions become unavailable. Also, by Remark 1, it is easy to convert $\mathscr{C}$ to $A(\mathscr{C})$ and vice versa. Therefore, we patch a simple controller $\mathscr{C}$ by modifying its corresponding FTS $A(\mathscr{C})$ and then transferring $A(\mathscr{C})$ to $\mathscr{C}$. Several operations applied to the FTS $A(\mathscr{C})$ are defined in Table 1, with which we are ready to modify simple controllers resulting from (13) and (14), i.e. $\text{Pre}^{T,U}_{\exists,\forall}(V)$ and $\text{Inv}^{D,G}_{\exists}(Z,B)$.

Suppose that W and $\mathscr{C}$ are the winning set and controller resulting from $\text{Pre}^{T,U}_{\exists,\forall}(V)$. Given $\widehat{V} = V - \Delta V$, we want to modify W and $\mathscr{C}$ to be the winning set and controller resulting from $\text{Pre}^{\widehat{T},\widehat{U}}_{\exists,\forall}(\widehat{V})$. Intuitively if we can find all the invalid transitions (s_1,u,s_2) in $A(\mathscr{C})$ due to $u \in U_d$ or $s_2 \in \Delta V$, and remove them from $A(\mathscr{C})$, that would result in a controller that works for the new problem setting, based on which, the patching operator for (13) is:

$$[\widehat{W},\widehat{\mathscr{C}}] = \overline{\text{Pre}}^{T,U}_{\exists,\forall}(\mathscr{C},\Delta V, U_d)$$
$$= \begin{cases} T_0 = A(\mathscr{C}) \backslash (*, U_d, *)_{\rightarrow_{A(\mathscr{C})}} \\ E = \widehat{\text{Pre}}^{T_0}_{\exists,\exists}(\Delta V) \\ A(\widehat{\mathscr{C}}) = T_0 \backslash (E,*)_{\rightarrow_{T_0}} \\ \widehat{W} = \{s_1 : \exists u, \exists s_2 \ s.t. \ (s_1,u,s_2) \in \rightarrow_{A(\widehat{\mathscr{C}})}\}, \end{cases} \tag{2}$$

where $\widehat{\mathscr{C}}$ is converted from $A(\widehat{\mathscr{C}})$. $\widehat{W}$ and $\widehat{\mathscr{C}}$ are the modified winning set and controller.

Theorem 3. $\widehat{W}$ and $\widehat{\mathscr{C}}$ returned by (2) are the same as the outputs resulting from $\text{Pre}^{\widehat{T},\widehat{U}}_{\exists,\forall}(\widehat{V})$ in (13).

Proof: Suppose that $\text{Pre}^{\widehat{T},\widehat{U}}_{\exists,\forall}(\widehat{V})$ returns W_t and $\mathscr{C}_t$. It is enough to show that $A(\widehat{\mathscr{C}}) = A(\mathscr{C}_t)$, i.e. $\rightarrow_{A(\widehat{\mathscr{C}})} = \rightarrow_{A(\mathscr{C}_t)}$. By (13) and Definition 7, it is obvious that $\rightarrow_{A(\mathscr{C}_t)} \subseteq \rightarrow_{A(\mathscr{C})}$ for $\widehat{V} \subseteq V$ and $\widehat{U} \subseteq U$. The transitions in $S = (*,U_d,*)_{\rightarrow_{A(\mathscr{C})}} \cup (E,*)_{\rightarrow_{T_0}}$ are either under actions that may push the system to ΔV or under actions in U_d, so $\rightarrow_{A(\mathscr{C}_t)}$ and S are disjoint. $\rightarrow_{A(\widehat{\mathscr{C}})} = \rightarrow_{A(\mathscr{C})} - S$. Thus $\rightarrow_{A(\mathscr{C}_t)} \subseteq \rightarrow_{A(\widehat{\mathscr{C}})}$. By (2), transitions in $A(\widehat{\mathscr{C}})$ can only take actions in $\widehat{U}$ and transit to states in $\widehat{V}$, which implies that $\rightarrow_{A(\widehat{\mathscr{C}})} \subseteq \rightarrow_{A(\mathscr{C}_t)}$. ■

Suppose that Y and $\mathscr{C}$ are the winning set and controller resulting from $\text{Inv}^{D,G}_{\exists}(Z,B)$. If $D \cap U_d \neq \emptyset$, the winning set is empty by the definition of progress group (see[12]). Given $\widehat{Z} \subseteq Z$ and D such that $D \cap U_d = \emptyset$, we want to modify Y and $\mathscr{C}$ to be the winning set and controller for $\text{Inv}^{D,G}_{\exists}(\widehat{Z},B)$. The modified winning set $\widehat{Y}$ is contained by $Y \cup (\Delta Z \cap G \cap B)$. Also, $\text{Inv}^{D,G}_{\exists}(Z,B)$ computation in (14) is a contraction algorithm in the sense that the sequence $\{Y_k\}_{k=1}^{\infty}$ is monotonically decreasing in set inclusion sense and its limit is the winning set. Combining these two facts, we can warm-start the contraction algorithm in (14) with $Y_0 = Y \cup (\Delta Z \cap G \cap B)$ instead of Q. The patching operator is:

$$[\widehat{Y},\widehat{\mathscr{C}}] = \overline{Inv}_{\exists}^{D,G}(Y,\mathscr{C},Z,\widehat{Z}) = \begin{cases} Y_0 = Y \cup (\Delta Z \cap G \cap B) \\ \Delta Y_0 = Q - (Y_0 \cup \widehat{Z}) \\ T_0 = A(\mathscr{C}) \cup E_0 \\ k = 0 \\ repeat: \\ \quad \Delta Y_{k+1} = \Delta Y_k \cup \mathrm{Pre}_{\forall,\exists}^{T_k,D}(\Delta Y_k) \end{cases} \begin{cases} \quad T_{k+1} = T_k \backslash (\Delta Y_{k+1},*,*)_{\rightarrow_{T_k}} \\ \quad Y_{k+1} = Y_k - \Delta Y_{k+1} \\ \quad k = k+1 \\ until\ Y_k = Y_{k-1} \\ \widehat{Y} = Y_k,\ [-,\widehat{\mathscr{C}}] = \mathrm{Pre}_{\exists,\forall}^{T_k,D}(\widehat{Y} \cup \widehat{Z}), \end{cases} \tag{3}$$

where $\Delta Z = Z - \widehat{Z}$, $E_0 = \{(s_1,a,s_2) \in \rightarrow_T : s_1 \in Y_0, a \in D, s_2 \in Q\}$, $\mathrm{Pre}_{\forall,\exists}^{T_k,D}(\Delta Y_k) = \{q_1 : \forall u \in D, \exists q_2 \in \Delta Y_k, s.t.\ (q_1,u,q_2) \in \rightarrow_{T_k}\}$. $\widehat{Y}$ is the modified winning set, and the controller $\widehat{\mathscr{C}}$ can be computed from T_k.

In regards to the patching operator in (3), we have the following result, proof of which is given in Appendix B:

Theorem 4. $\widehat{Y}$ and $\widehat{\mathscr{C}}$ returned by (3) are the same as the outputs resulting from $\mathrm{Inv}_{\exists}^{D,G}(\widehat{Z},B)$ in (14).

Given a simple controller $\mathscr{C}$ and new synthesis settings, both (2) and (3) try to find the modified winning sets inside $A(\mathscr{C})$ (or $A(\mathscr{C}) \cup E_0$). However, if we re-synthesize the new simple controllers from scratch, (13) and (14) start to find the winning sets in the whole AFTS T, which in general needs more computation cost, demonstrated by the simulation results in Section 4. However, in the worst case, there might not be any computational gains.

3.2 Patching General Controllers

In the section we are going to patch all the non-leaf nodes, i.e the controllers resulting from (10), (11), (12), and the root node, i.e. the controller result from (9), and finally Theorem 8 shows that the patching algorithm for (9) offers a solution to the Problem 1 in Section 2.5.

Assume that Z and $\mathscr{C} = (\mathscr{V},\mathscr{K},x)$ are the winning set and controller resulting from $\mathrm{PGPre}_{\exists,\forall}^{T}(Z,B)$. Given $\widehat{Z} \subseteq Z$, we want to modify Z and $\mathscr{C}$ to be the winning set and controller for $\mathrm{PGPre}_{\exists,\forall}^{\widehat{T}}(\widehat{Z},B)$. The patching operator is:

$$[\widehat{Z}',\widehat{\mathscr{C}}] = \overline{\mathrm{PGPre}}_{\exists,\forall}^{T}(\mathscr{C},Z,\widehat{Z},U_d) = \begin{cases} Z' = Z,\ \widehat{Z}' = \widehat{Z},\ \widehat{\mathscr{V}} = \{\},\ \widehat{\mathscr{K}} = \{\},\ k = 1 \\ for\ D \in 2^U: \\ \quad for\ G \in G(D): \\ \quad\quad if\ U_d \cap D = \emptyset: \\ \quad\quad\quad [\widehat{\mathscr{V}}(k),\widehat{\mathscr{K}}(k)] = \overline{Inv}_{\exists}^{D,G}(\mathscr{V}(k),\mathscr{K}(k),Z',\widehat{Z}') \end{cases} \begin{cases} \quad\quad else:\ \widehat{\mathscr{V}}(k) = \emptyset, \widehat{\mathscr{K}}(k) = \emptyset \\ \quad\quad Z' = Z' \cup \mathscr{V}(k),\ \widehat{Z}' = \widehat{Z}' \cup \widehat{\mathscr{V}}(k) \\ \quad\quad k = k+1 \\ Remove\ all\ \emptyset\ in\ \widehat{\mathscr{V}}\ and\ \widehat{\mathscr{K}},\ \widehat{\mathscr{C}} = (\widehat{\mathscr{V}},\widehat{\mathscr{K}},x). \end{cases} \tag{4}$$

Theorem 5. $\widehat{Z}'$ and $\widehat{\mathscr{C}}$ returned by (4) are the same as outputs resulting from $\mathrm{PGPre}_{\exists,\forall}^{\widehat{T}}(\widehat{Z},B)$ in (12).

Proof: By theorem 4, the results of (4) remain the same if we replace $\overline{Inv}_{\exists}^{D,G}(\mathscr{V}(k),\mathscr{K}(k),Z',\widehat{Z}')$ in (4) with $Inv_{\exists}^{D,G}(\widehat{Z}')$. After the replacement, the algorithm (4) is the same as the algorithm (12). Therefore, their outputs must be the same. ■

Assume that X_∞ and $\mathscr{C} = (\mathscr{V}, \mathscr{K}, x)$ are the winning set and controller resulting from $\mathrm{Win}^{T}_{\exists,\forall}(B\ \mathbf{U}\ Z)$. Assume $|\mathscr{K}| = 2n$, i.e. (11) converges in n steps. Then, given $\widehat{Z} \subseteq Z$, to get the winning set and controller for $\mathrm{Win}^{\widehat{T}}_{\exists,\forall}(B\ \mathbf{U}\ \widehat{Z})$, the patching operator is:

$$[\widehat{X}_\infty, \widehat{\mathscr{C}}] = \overline{\mathrm{Win}}^{T}_{\exists,\forall,(B\ \mathbf{U}\ Z)}(\mathscr{C}, Z, \widehat{Z}, U_d)$$

$$= \begin{cases} X_0 = \emptyset, \widehat{X}_0 = \emptyset, \Delta X = \emptyset,\ \widehat{\mathscr{V}} = \{\},\ \widehat{\mathscr{K}} = \{\} \\ k = 0 \\ repeat: \\ \quad [\widehat{\mathscr{V}}(2k+1), \widehat{\mathscr{K}}(2k+1)] = \\ \quad\quad \overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathscr{K}(2k+1), X_k - \widehat{X}_k, U_d) \\ \quad E_k = Z \cup (B \cap \mathscr{V}(2k+1)) \\ \quad \widehat{E}_k = Z \cup (B \cap \widehat{\mathscr{V}}(2k+1)) \\ \quad [\widehat{\mathscr{V}}(2k+2), \widehat{\mathscr{K}}(2k+2)] = \\ \quad\quad \overline{\mathrm{PGPre}}^{T}_{\exists,\forall}(\mathscr{K}(2k+2), E_k, \widehat{E}_k, U_d) \\ \quad X_{k+1} = Z \cup (B \cap \mathscr{V}(2k+1)) \cup \mathscr{V}(2k+2) \\ \quad \widehat{X}_{k+1} = \widehat{Z} \cup (B \cap \widehat{\mathscr{V}}(2k+1)) \cup \widehat{\mathscr{V}}(2k+2) \end{cases} \begin{cases} \quad k = k+1 \\ until\ k = n\ or\ \widehat{X}_k = \widehat{X}_{k-1} \\ while\ \widehat{X}_k \neq \widehat{X}_{k-1}: \\ \quad [\widehat{\mathscr{V}}(2k+1), \widehat{\mathscr{K}}(2k+1)] = \\ \quad\quad \overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathscr{K}(2n-1), X_n - \widehat{X}_k, U_d) \\ \quad \widehat{E}_k = Z \cup (B \cap \widehat{\mathscr{V}}(2k+1)) \\ \quad [\widehat{\mathscr{V}}(2k+2), \widehat{\mathscr{K}}(2k+2)] = \\ \quad\quad \overline{\mathrm{PGPre}}^{T}_{\exists,\forall}(\mathscr{K}(2n), E_n, \widehat{E}_k, U_d) \\ \quad \widehat{X}_{k+1} = \widehat{Z} \cup (B \cap \widehat{\mathscr{V}}(2k+1)) \cup \widehat{\mathscr{V}}(2k+2) \\ \quad k = k+1 \\ \widehat{X}_\infty = \widehat{X}_k,\ \widehat{\mathscr{C}} = (\widehat{\mathscr{V}}, \widehat{\mathscr{K}}, x). \end{cases} \tag{5}$$

In the algorithm (5), for $k < n$, we patch the $(2k+1)$th and $(2k+2)$th existing sub-controllers in $\mathscr{K}$. If the winning set does not converge in n iterations, for $k \geq n$, we duplicate the last two existing sub-controllers to the tail of $\mathscr{K}$, and patch them to enlarge the winning set until convergence.

Theorem 6. $\widehat{X}_\infty$ and $\widehat{\mathscr{C}}$ returned by (5) are the same as outputs resulting from $\mathrm{Win}^{\widehat{T}}_{\exists,\forall}(B\ \mathbf{U}\ \widehat{Z})$ in (11).

Proof: Similar to the proof of Theorem 5, we can replace all the $\overline{Pre}^{T,U}_{\exists,\forall}$ and $\overline{PGPre}^{T}_{\exists,\forall}$ terms with $\overline{Pre}^{\widehat{T},\widehat{U}}_{\exists,\forall}(\widehat{X}_k)$ and $PGPre^{\widehat{T}}_{\exists,\forall}(\widehat{E}_k, B)$. Theorem 3 and 5 guarantee that the replacements are equivalent, but we need to verify: (i) $\widehat{X}_k \subseteq X_k$ and $\widehat{E}_k \subseteq E_k$ for $k < n$, and (ii) $\widehat{X}_k \subseteq X_n$ and $\widehat{E}_k \subseteq E_n$ for $k \geq n$. Both (i) and (ii) can be easily checked using induction argument (with the base case $\widehat{X}_0 \subseteq X_0$ and $\widehat{E}_1 \subseteq E_1$) and Theorem 2. After the equivalent replacements, algorithm (5) and algorithm (11) become the same. Thus their outputs must be the same. ■

Assume that W and $\mathscr{C} = (\mathscr{V}, \mathscr{K}, x)$ are the winning set and controller resulting from $\mathrm{Win}^{T}_{\exists,\forall}((B\ \mathbf{U}\ Z) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))$. Given that $\widehat{Z} \subseteq Z$ and $\widehat{U} \subseteq U$, patch W and $\mathscr{C}$ for $\mathrm{Win}^{T}_{\exists,\forall}((B\ \mathbf{U}\ \widehat{Z}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))$. The patching operator for (10) is:

$$[\widehat{W}_\infty, \widehat{\mathscr{C}}] = \overline{\mathrm{Win}}^{T}_{\exists,\forall(\psi)}(\mathscr{C}, Z, \widehat{Z}, U_d)$$

$$= \begin{cases} W_0 = \mathscr{V}(1), \widehat{\mathscr{V}} = \mathscr{V}, \widehat{\mathscr{K}} = \mathscr{K} \\ Z^i_\infty = Z \cup (B \cap R^i \cap \mathrm{Pre}^{T,U}_{\exists,\forall}(W_0)) \\ \widehat{Z}^i_0 = \widehat{Z} \cup (B \cap R^i \cap \mathrm{Pre}^{T,U-U_d}_{\exists,\forall}(W_0)) \\ [X^i_0, \widehat{\mathscr{K}}(i)] = \overline{\mathrm{Win}}^{T}_{\exists,\forall,(B\mathbf{U}Z)}(\mathscr{K}(i), Z^i_\infty, \widehat{Z}^i_0, U_d) \\ \widehat{W}_0 = \bigcap_{i\in I} X^i_0 \\ k = 0 \end{cases} \begin{cases} repeat: \\ \quad \widehat{Z}^i_{k+1} = \widehat{Z} \cup (B \cap R^i \cap \mathrm{Pre}^{T,U-U_d}_{\exists,\forall}(\widehat{W}_k)) \\ \quad [X^i_{k+1}, \widehat{\mathscr{K}}(i)] = \overline{\mathrm{Win}}^{T}_{\exists,\forall,(B\mathbf{U}Z)}(\widehat{\mathscr{K}}(i), \widehat{Z}^i_k, \widehat{Z}^i_{k+1}, U_d) \\ \quad \widehat{W}_{k+1} = \bigcap_{i\in I} X^i_{k+1} \\ \quad k = k+1 \\ until\ \widehat{W}_k = \widehat{W}_{k-1} \\ \widehat{W}_\infty = \widehat{W}_k,\ \widehat{\mathscr{C}} = (\widehat{\mathscr{V}}, \widehat{\mathscr{K}}, x). \end{cases} \tag{6}$$

Theorem 7. $\widehat{W}_\infty$ and $\widehat{\mathscr{C}}$ returned by (5) are the same as outputs resulting from $\mathrm{Win}^{\widehat{T}}_{\exists,\forall}((B\ \mathbf{U}\ \widehat{Z}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))$ in (10).

Proof: Assume that W_t is the winning set resulting from $\mathrm{Win}^{\widehat{T}}_{\exists,\forall}((B\,\mathbf{U}\,\widehat{Z})\vee\Box(B\wedge(\bigwedge_{i\in I}\Diamond R^i))$. Similar to the proof of Theorem 6, we can replace all the $\overline{Win}^{T}_{\exists,\forall}$ terms in (6) with $Win^{\widehat{T}}_{\exists,\forall}(B\mathbf{U}\widehat{Z}^i_k)$, as long as (i) $\widehat{Z}^i_0\subseteq Z^i_\infty$ and (ii) $\widehat{Z}_{k+1}\subseteq\widehat{Z}_k$ are true. (i) is trivial. For (ii), it is enough to show that $\widehat{W}_{k+1}\subseteq\widehat{W}_k$, which can be proven by induction. The base case: Since (i) is true, by Theorem 6, we have $X^i_0=Win^{\widehat{T}}_{\exists,\forall}(B\mathbf{U}\widehat{Z})$. Then $\widehat{W}_0=\bigcap_{i\in I}\mathrm{Win}^{\widehat{T}}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_0)\subseteq\bigcap_{i\in I}\mathrm{Win}^{T}_{\exists,\forall}(B\,\mathbf{U}\,Z^i_\infty)=W_0$ by Theorem 2. Assume that $\widehat{W}_{k+1}\subseteq\widehat{W}_k$. Then $\widehat{Z}^i_{k+1}\subseteq\widehat{Z}^i_k$ and $\widehat{W}_{k+1}=\bigcap_{i\in I}\mathrm{Win}^{\widehat{T}}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_{k+1})\subseteq\bigcap_{i\in I}\mathrm{Win}^{\widehat{T}}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_k)=\widehat{W}_k$. Therefore by induction argument $\widehat{W}_{k+1}\subseteq\widehat{W}_k$, and $\widehat{Z}_{k+1}\subseteq\widehat{Z}_k$ for all k. Then (ii) is true. After the replacements, (6) and (10) are the same except that the contraction of W_k starts from the existing winning set W instead of Q. We can show that $W_t\subseteq\widehat{W}_k$ for all k by induction (base case: $W_t\subseteq W_0$ by Theorem 2). Also W_t is the largest fixed-point in Q by definition. So both (6) and (10) will converge to the same fixed-point W_t. ■

Finally we are ready to patch the controller resulting from (9) for specification in the form of (1).

Assume that W and $\mathscr{C}=(\mathscr{V},\mathscr{K},x)$ are the winning set and controller resulting from the operator $\mathrm{Win}^{T}_{\exists,\forall}\left(\Box A\wedge\Diamond\Box B\wedge\left(\bigwedge_{i\in I}\Box\Diamond R^i\right)\right)$. Assuming that $|\mathscr{K}|=n$. For patching purpose, we need some extra information: the lists of winning sets and controllers returned by $\mathrm{Pre}^{T,U}_{\exists,\forall}(V_k)$ in (9), i.e. $[\mathscr{V}_1(k),\mathscr{K}_1(k)]=\mathrm{Pre}^{T,U}_{\exists,\forall}(V_k)$; the lists of winning sets and controllers returned by $\mathrm{PGPre}^{T}_{\exists,\forall}(V_k,B)$ in (9), i.e. $[\mathscr{V}_2(k),\mathscr{K}_2(k)]=\mathrm{PGPre}^{T}_{\exists,\forall}(V_k,B)$ $(k=1,...,n)$; the controller $\mathscr{C}_{Inv}$ returned by $\mathrm{Win}^{T}_{\exists,\forall}((A\mathbf{U}\emptyset)\vee\Box(A\wedge\Diamond Q))$ in the first line of (9). To restrict action space to $\widehat{U}=U-U_d$, the patching operator for (9) is:

$$[\widehat{V}_\infty,\widehat{\mathscr{C}}]=\overline{\mathrm{Win}}^{T}_{\exists,\forall,(\phi)}(\mathscr{C},\mathscr{C}_{Inv},\mathscr{V}_1,\mathscr{K}_1,\mathscr{V}_2,\mathscr{K}_2,U_d)$$

$$=\left\{\begin{array}{l}
\widehat{V}_{\mathrm{inv}}=\overline{\mathrm{Win}}^{T}_{\exists,\forall,(\psi)}(\mathscr{C}_{Inv},\emptyset,\emptyset,U_d)\\
\textit{Restrict synthesis to }\widehat{V}_{\mathrm{inv}}\\
\widehat{V}_0=\emptyset,\ \mathscr{V}(0)=\emptyset,\ \widehat{\mathscr{V}}=\{\},\ \widehat{\mathscr{K}}=\{\}\\
k=0\\
\textit{repeat}:\\
\quad\widehat{Z}^1_{k+1}=\overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathscr{K}_1(k+1),\mathscr{V}(k)-\widehat{V}_k,U_d)\\
\quad\widehat{Z}^2_{k+1}=\overline{\mathrm{PGPre}}^{T}_{\exists,\forall}(\mathscr{K}_2(k+1),\mathscr{V}(k),\widehat{V}_k,U_d)\\
\quad Z_{k+1}=\mathscr{V}_1(k+1)\cup\mathscr{V}_2(k+1),\ \widehat{Z}_{k+1}=\widehat{Z}^1_{k+1}\bigcup\widehat{Z}^2_{k+1}\\
\quad[\widehat{V}_{k+1},\widehat{\mathscr{K}}(k+1)]=\\
\qquad\overline{\mathrm{Win}}^{T}_{\exists,\forall,(\psi)}(\mathscr{K}(k+1),Z_{k+1},\widehat{Z}_{k+1},U_d)\\
\quad\widehat{\mathscr{V}}(k+1)=\widehat{V}_{k+1}
\end{array}\right.
\left\{\begin{array}{l}
\quad k=k+1\\
\textit{until }k=n\textit{ or }\widehat{V}_k=\widehat{V}_{k-1}\\
\textit{while }\widehat{V}_k\neq\widehat{V}_{k-1}:\\
\quad\widehat{Z}^1_{k+1}=\overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathscr{K}_1(n),\mathscr{V}(n-1)-\widehat{V}_k,U_d)\\
\quad\widehat{Z}^2_{k+1}=\overline{\mathrm{PGPre}}^{T}_{\exists,\forall}(\mathscr{K}_2(n),\mathscr{V}(n-1),\widehat{V}_k,U_d)\\
\quad\widehat{Z}_{k+1}=\widehat{Z}^1_{k+1}\bigcup\widehat{Z}^2_{k+1}\\
\quad[\widehat{V}_{k+1},\widehat{\mathscr{K}}(k+1)]=\\
\qquad\overline{\mathrm{Win}}^{T}_{\exists,\forall,(\psi)}(\mathscr{K}(n),Z_n,\widehat{Z}_{k+1},U_d)\\
\quad\widehat{\mathscr{V}}(k+1)=\widehat{V}_{k+1}\\
\quad k=k+1\\
\widehat{V}_\infty=\widehat{V}_k,\widehat{\mathscr{C}}=(\widehat{\mathscr{V}},\widehat{\mathscr{K}},x).
\end{array}\right.\tag{7}$$

The same as (5), the patching algorithm firstly patches the existing sub-controllers. If the winning set does not converge in n iterations, the last existing controller is duplicated to the tail of $\widehat{\mathscr{K}}$ and patched to enlarge the winning set until it converges.

Theorem 8. $\widehat{V}_\infty$ and $\widehat{\mathscr{C}}$ returned by (7) are the same as outputs resulting from (9) that is

$$\mathrm{Win}^{\widehat{T}}_{\exists,\forall}\left(\Box A\wedge\Diamond\Box B\wedge\left(\bigwedge_{i\in I}\Box\Diamond R^i\right)\right).$$

.

Proof: Similar to the previous proofs, we want to replace all the $\overline{Pre}^{T,U}_{\exists,\forall}$, $\overline{PGPre}^{T}_{\exists,\forall}$ and $\overline{Win}^{T}_{\exists,\forall,(\Psi)}$ terms in (7) with $Pre^{\widehat{T},\widehat{U}}_{\exists,\forall}(\widehat{V}_k)$, $PGPre^{\widehat{T}}_{\exists,\forall}(\widehat{V}_k,Q)$ and $Win^{\widehat{T}}_{\exists,\forall}((B\,\mathbf{U}\,\widehat{Z}_{k+1})\vee\Box(B\wedge(\bigwedge_{i\in I}\Diamond R^i))))$, as long as

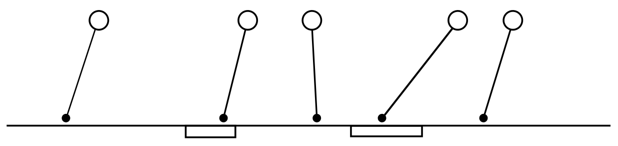


Figure 3: Simple model of one-legged 1D robots with point foot walking on a ground with holes.

(i) $\widehat{V}_k \subseteq \mathscr{V}(k)$ and $\widehat{Z}_{k+1} \subseteq Z_{k+1}$ for $k < n$ and (ii) $\widehat{V}_k \subseteq \mathscr{V}(n-1)$ and $\widehat{Z}_{k+1} \subseteq Z_n$ for $k \geq n$. Both (i) and (ii) can be proven by induction using Theorems 2, 3, 5 and 7. (First prove (i) and then use the case where $k = n-1$ in (i) to be the base case in the induction argument for (ii)). Also, in the first line of (7) we use $\overline{Win}^{T}_{\exists,\forall,(\Psi)}$ to compute $\widehat{V}_{inv}$, which can can be replaced with $Win^{\widehat{T}}_{\exists,\forall}((A\mathbf{U}\emptyset) \vee \Box(A \wedge \Diamond Q))$ since $\emptyset \subseteq \emptyset$. After the replacements, (7) and (9) become exactly the same, which implies that their outputs must be the same. ■

Theorem 8 shows that our patching method gives the same results as re-synthesizing from scratch via (9), and furthermore, since the algorithm (9) is sound and complete, our method is sound and complete, therefore it offers a solution to the Problem 1 stated in Section 2.5.

4 EXAMPLE

4.1 Simple Transition System

For the transition system in Figure 1, take the controller we get in Example 1 and $U_d = \{e\}$ as input of the patching algorithm (7). The outputs are the modified winning set $\widehat{W} = \{s_1, s_2, s_3\}$ and controller $\widehat{\mathscr{C}} = (\{\widehat{W}\}, \{\widehat{\mathscr{C}}^0_1\}, x = 1)$, where the descendant controllers are:
$\widehat{\mathscr{C}}^0_1 = (\{\widehat{W}, \{s_1\}, \{s_2, s_3\}\}, \{\widehat{\mathscr{C}}^1_1, \widehat{\mathscr{C}}^1_2\}, x^0)$; $\widehat{\mathscr{C}}^1_1 = (\{\{s_1, s_2\}, W, W\}, \{\widehat{\mathscr{C}}^{2,0}_i\}^3_{i=1}, x^1_0)$;
$\widehat{\mathscr{C}}^1_2 = (\{\{s_2, s_3\}, W, W\}, \{\widehat{\mathscr{C}}^{2,1}_i\}^3_{i=1}, x^1_1)$; $\widehat{\mathscr{C}}^{2,0}_1 = \{(s_1, \{a\}), (s_2, \{c\})\}$;
$\widehat{\mathscr{C}}^{2,0}_2 = \{(s_1, \{a, b\}), (s_2, \{c\}), (s_3, \{g\})\}$; $\widehat{\mathscr{C}}^{2,0}_3 = \{(s_1, \{a, b\}), (s_2, \{c, f\}), (s_3, \{g, h\})\}$;
$\widehat{\mathscr{C}}^{2,1}_1 = \{(s_2, \{f\}), (s_3, \{h\})\}$; $\widehat{\mathscr{C}}^{2,1}_2 = \{(s_1, \{b\}), (s_2, \{f\}), (s_3, \{g, h\})\}$;
$\widehat{\mathscr{C}}^{2,1}_3 = \{(s_1, \{a, b\}), (s_2, \{c, f\}), (s_3, \{g, h\})\}$;

Let the system start from s_1 and initialize internal variables as $(1, 1, 1, 1)$. Only one trajectory is available under control of $\widehat{\mathscr{C}}$, i.e. $s_1, s_2, s_3, s_2, s_1, \ldots$, where the sequence of actions is $b, f, g, c, \ldots$ according to the controller execution rules in Definition 6. The trajectory does not visit s_4, for the system is not able to come back to A from s_4 after action e is removed.

4.2 Case study: 1D Walking Robot

We consider a model of one-legged 1D walking robot moving on a straight line, called the Linear Inverted Pendulum Model (LIPM, see [6]). This is a very simplified model of dynamics of a walking robot, but it is used by many algorithms of bipedal walking control (e.g. [5]). It consists in a point-mass robot with a massless leg moving along a line at a constant height above the ground. We do not restrict the extension of the leg nor the velocity of the leg motion (the replacement of the foot is instantaneous). The dynamics of the model are:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ g/h_0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ -g/h_0 \end{bmatrix} u \tag{8}$$

Table 2: The execution time of re-synthesis from scratch (row 3) and patching method (row 4) under multiple action profiles. The first row is the set of available actions. The second row is the percentage of transitions left after U_d is disabled.

U_d	$[1]$	$[1:5]$	$[1:10]$	$[1:15]$	$[1:20]$	$[1:25]$
$\exists$trans	100%	96.57%	81.22%	60.55%	39.67%	19.00%
$t_{syn}(s)$	10.9	24.5	36.4	32.7	25.5	14.6
$t_{pat}(s)$	1.3	5.7	7.7	7.3	5.9	4.8

Table 3: The average execution time for re-synthesizing from scratch (row 2), re-synthesizing from the existing winning set, i.e. restrict the state space of the AFTS to be the existing winning set and re-run (9) (row 3) and our patching method (row 4) under random action profiles. The first row is number of unavailable actions ($\sharp U_d$). For each number, choose 10 random sets of unavailable actions.

$\sharp U_d$	1	5	10	15	20	25
$t_{syn}(s)$	18.1	17.5	16.8	22.8	33.0	29.5
$t_{ws}(s)$	4.4	5.5	6.4	10.5	16.8	22.3
$t_{pat}(s)$	1.2	1.6	1.9	2.8	4.3	5.8

where x is the horizontal position of the center of mass (CoM) of the robot, v is the velocity of CoM and u is the horizontal position of the foot. that the robot foot will step on.

We consider the state space $Q = [-2.5, 2.5] \times [-4, 4]$ with action space $U = [-3.5, 3.5]$. Discretize Q and U uniformly with grid size 0.1 and 0.2, and compute the transitions between discretized grids over-approximately using the method described in [8, 16]. Finally we get a AFTS T with states indexed $[1:4000]$, actions $[1:35]$ and 187594 valid transitions, which is the abstraction of the walking robot.

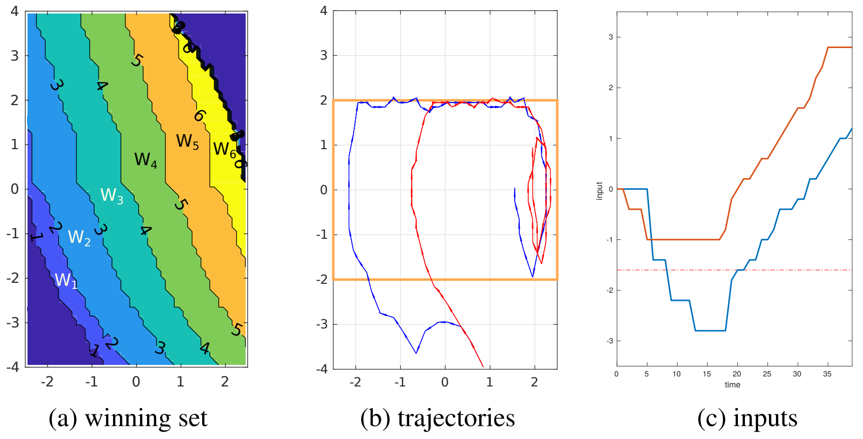
The specification for the control synthesis is $\Box Q \wedge \Diamond\Box B$, where $B = [-2.5, 2.5] \times [-2, 2]$. It says that the CoM of the robot should always stay within $[-2.5, 2.5]$ with velocity lower than 2 after finite time from beginning. Given the specification, the winning set and controller are computed via (9) (taking $A = Q, B = B, C^1 = Q$).

Now imagine that some holes on the ground are detected, as shown in Figure 3, where the robot should avoid stepping. Therefore some actions need to be disabled. Once we determine which actions will be affected, we can put them in U_d and patch the existing controllers for the new action profiles.

The experiment environment is MATLAB R2017a with CPU Intel Core i7-6820 HQ.

We choose unavailable actions $U_d = [1], [1:5], ..., [1:25]$ for the walking robot abstraction and compute the controllers via fixed-point operator (9) and the patching operator (7) respectively. The experiment results in Table 2 make a comparison between the time of synthesizing from scratch (t_{syn}) with the time of patching existing controllers (t_{pat}), which shows that our patching methods can shorten the synthesis time significantly. Figure 4a shows the winning sets for each U_d, which shrink to the right part in the state space as U_d (region of holes) grows.

To further show the time efficiency, we randomly choose U_d with size $n = 1, 5, ..., 25$. By Theorem 2, the winning set after U_d is removed is contained by the existing one, so we can restrict the state space for the AFTS T to be the existing winning set and synthesize over this new abstraction via (9). Theoretically it saves computation cost, for the new AFTS is smaller. We call it naive warm-starting method. Table 3 compares the average time used by naive warm-starting (row 3) with our patching method (row 4) and re-synthesizing from scratch (row 2) for each n. The time for our patching algorithm is up to 20% of the time for re-synthesizing and up to 30% of the time for naive warm-starting on average. Also, as shown



(a) winning set (b) trajectories (c) inputs

Figure 4: (a) Six regions with different colors are labeled as $W_1, W_2, ..., W_6$. The winning sets under action profiles $U_d = [1], [1 : 5], [1 : 10], ..., [1 : 25]$ are the regions corresponding to $\bigcup_{i=1}^{6} W_i$, $\bigcup_{i=2}^{6} W_i, \bigcup_{i=3}^{6} W_i, ...,$ W_6 respectively. (b) Trajectories with initial state $(0.85, -3.95)$ under $U_d = \emptyset$ (blue) and $[1 : 10]$ (red). The region inside orange box is the target set B. (c) Inputs over time under $U_d = \emptyset$ (blue) and $[1 : 10]$ (red) for trajectories in (b). The red dash-point line indicates value corresponding to the input indexed by $u = 10$.

in the table, the more actions are unavailable, the more time the patching algorithm needs. This can be attributed to the fact that when more actions are unavailable, the fixed point is likely to change more.

Finally, to show that formal guarantees are satisfied after patching, simulations are run for controllers before and after patching for $U_d = [1 : 10]$. The initial state is $s_0 = 34$. Figures 4b and 4c show the trajectories and the inputs used within 60 time steps. Both trajectories go into our target region B, indicated by the orange box. The outputs from patched controller are always above the dash line where $u = 10$, due to the unavailability of actions $[1 : 10]$.

For practical applications in robot control, if we have the controller for the case that no constraint exists and know all the possible profiles of actions (all the possible constraints on the surface) for a known environment, the patching algorithm can generate the corresponding controllers for those action profiles very quickly.

5 CONCLUSION

In this paper, we proposed an implementation (data structure) for a controller synthesized by fixed-point based control synthesis techniques in [12]. For an existing controller with such a structure, a patching algorithm was developed to modify it for the case that some actions in the original problem setting become unavailable. Furthermore we proved that the winning set resulting from our patching algorithm was exactly the same as the winning set resulting from synthesizing a new controller from scratch.

We illustrated the efficiency of our method on a example of walking robot. The controller was synthesized for the robot described by a Linear Inverted Pendulum Model with hole constraints on the surface. We first synthesized a controller for a smooth surface without holes, and then patched it using our method for the cases that holes existed. Under the same specification, the time used by our method was only $1.4\% - 7\%$ of the time used for synthesis from scratch and less than 3% on average.

As future work, it would be interesting to consider cases where the modified action set is not a strict subset of the original action set.

References

[1] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT press.

[2] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli & Yaniv Saar (2012): *Synthesis of reactive (1) designs*. Journal of Computer and System Sciences 78(3), pp. 911–938.

[3] Rüdiger Ehlers (2011): *Generalized Rabin (1) synthesis with applications to robust system synthesis*. In: NASA Formal Methods Symposium, Springer, pp. 101–115.

[4] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis & Moshe Y Vardi (2017): *Supervisory control and reactive synthesis: a comparative introduction*. Discrete Event Dynamic Systems 27(2), pp. 209–260.

[5] Johannes Englsberger, Christian Ott, Máximo A Roa, Alin Albu-Schäffer & Gerhard Hirzinger (2011): *Bipedal walking control based on capture point dynamics*. In: Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, IEEE, pp. 4420–4427.

[6] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kazuhito Yokoi & Hirohisa Hirukawa (2001): *The 3D Linear Inverted Pendulum Mode: A simple modeling for a biped walking pattern generation*. In: Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 239–246.

[7] Uri Klein & Amir Pnueli (2010): *Revisiting synthesis of GR (1) specifications*. In: Haifa Verification Conference, Springer, pp. 161–181.

[8] Jun Liu & Necmiye Ozay (2014): *Abstraction , Discretization , and Robustness in Temporal Logic Control of Dynamical Systems*. International Conferance on Hybrid Systems: Computation and Control (HSCC), pp. 293–302, doi:10.1145/2562059.2562137.

[9] Scott C. Livingston & Richard M. Murray (2014): *Hot-swapping robot task goals in reactive formal synthesis*. Proceedings of the IEEE Conference on Decision and Control, pp. 101–107, doi:10.1109/CDC.2014.7039366.

[10] Scott C Livingston, Pavithra Prabhakar, Alex B Jose & Richard M Murray (2013): *Patching task-level robot controllers based on a local μ-calculus formula*. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, IEEE, pp. 4588–4595.

[11] Shahar Maoz & Jan Oliver Ringert (2016): *On well-separation of GR (1) specifications*. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp. 362–372.

[12] Petter Nilsson, Necmiye Ozay & Jun Liu (2017): *Augmented finite transition systems as abstractions for control synthesis*. Discrete Event Dynamic Systems: Theory and Applications 27(2), pp. 301–340, doi:10.1007/s10626-017-0243-z. Available at `http://link.springer.com/10.1007/s10626-017-0243-z`.

[13] Amir Pnueli & Roni Rosner (1989): *On the synthesis of a reactive module*. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, pp. 179–190.

[14] Peter J Ramadge & W Murray Wonham (1987): *Supervisory control of a class of discrete event processes*. SIAM journal on control and optimization 25(1), pp. 206–230.

[15] Anne-Kathrin Schmuck, Thomas Moor & Rupak Majumdar (2017): *On the Relation between Reactive Synthesis and Supervisory Control of Non-Terminating Processes*. Technical Report, MPI-SWS.

[16] Fei Sun, Necmiye Ozay, Eric M Wolff, Jun Liu & Richard M Murray (2014): *Efficient control synthesis for augmented finite transition systems with an application to switching protocols*. In: Proceedings of the American Control Conference, pp. 3273–3280, doi:10.1109/ACC.2014.6859428.

[17] Eric M Wolff, Ufuk Topcu & Richard M Murray (2013): *Efficient reactive controller synthesis for a fragment of linear temporal logic*. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, IEEE, pp. 5033–5040.

$$[V_\infty, \mathscr{C}] = \mathrm{Win}^T_{\exists,\forall}\left(\Box A \wedge \Diamond\Box B \wedge \left(\bigwedge_{i\in I} \Box\Diamond R^i\right)\right)$$
$$= \begin{cases} V_{\mathrm{inv}} = \mathrm{Win}^T_{\exists,\forall}((A\mathbf{U}\emptyset) \vee \Box(A \wedge \Diamond Q)) \\ \text{Restrict synthesis to } V_{\mathrm{inv}} \\ V_0 = \emptyset,\ \mathscr{V} = \{\},\ \mathscr{K} = \{\},\ k = 0 \\ \mathit{repeat}: \\ \quad Z_{k+1} = \mathrm{Pre}^{T,U}_{\exists,\forall}(V_k) \bigcup \mathrm{PGPre}^T_{\exists,\forall}(V_k, Q) \\ \quad [V_{k+1}, \mathscr{C}_{k+1}] = \\ \quad \mathrm{Win}^T_{\exists,\forall}((B\,\mathbf{U}\,Z_{k+1}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))) \\ \quad \mathscr{V}(k+1) = V_{k+1},\ \mathscr{K}(k+1) = \mathscr{C}_{k+1} \\ \quad k = k+1 \\ \mathit{until}\ V_k = V_{k-1} \\ V_\infty = V_k,\ \mathscr{C} = (\mathscr{V}, \mathscr{K}, x) \end{cases} \tag{9}$$

$$[W_\infty, \mathscr{C}] = \mathrm{Win}^T_{\exists,\forall}((B\,\mathbf{U}\,Z) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i)))$$
$$= \begin{cases} W_0 = Q,\ \mathscr{V} = \{\},\ \mathscr{K} = \{\} \\ k = 0 \\ \mathit{repeat}: \\ \quad Z^i_{k+1} = Z \cup (B \cap R^i \cap \mathrm{Pre}^{T,U}_{\exists,\forall}(W_k)) \\ \quad [X^i, \mathscr{C}^i] = \mathrm{Win}^T_{\exists,\forall}(B\,\mathbf{U}\,Z^i_{k+1}), \forall i \in I \\ \quad W_{k+1} = \bigcap_{i\in I} X^i \\ \quad k = k+1 \\ \mathit{until}\ W_k = W_{k-1} \\ \mathscr{V}(1) = W_k, \\ \mathscr{V}(i+1) = B \cap R^i,\ \mathscr{K}(i) = \mathscr{C}^i, \forall i \in I \\ W_\infty = W_k,\ \mathscr{C} = (\mathscr{V}, \mathscr{K}, x) \end{cases} \tag{10}$$

$$[X_\infty, \mathscr{C}] = \mathrm{Win}^T_{\exists,\forall}(B\,\mathbf{U}\,Z) =$$
$$\begin{cases} X_0 = \emptyset,\ \mathscr{V} = \{\},\ \mathscr{K} = \{\},\ k = 0 \\ \mathit{repeat}: \\ \quad [V^1_k, C^1_k] = \mathrm{Pre}^{T,U}_{\exists,\forall}(X_k) \\ \quad [V^2_k, C^2_k] = \mathrm{PGPre}^T_{\exists,\forall}(Z \cup (B \cap V^1_k), B) \\ \quad X_{k+1} = Z \cup (B \cap V^1_k) \cup V^2_k \\ \quad \mathscr{V}(2k+1) = V^1_k, \mathscr{V}(2k+2) = V^2_k \\ \quad \mathscr{K}(2k+1) = C^1_k,\ \mathscr{K}(2k+2) = C^2_k \\ \quad k = k+1 \\ \mathit{until}\ X_k = X_{k-1} \\ X_\infty = X_k,\ \mathscr{C} = (\mathscr{V}, \mathscr{K}, x) \end{cases} \tag{11}$$

$$[Z', \mathscr{C}] = \mathrm{PGPre}^T_{\exists,\forall}(Z, B) =$$
$$\begin{cases} Z' = Z,\ \mathscr{V} = \{\},\ \mathscr{K} = \{\} \\ k = 1 \\ \mathit{for}\ D \in 2^U: \\ \quad \mathit{for}\ G \in G(D): \\ \quad\quad [V_k, \mathscr{C}_k] = \mathit{Inv}^{D,G}_{\exists}(Z', B) \\ \quad\quad Z' = Z' \cup V_k \\ \quad\quad \mathscr{V}(k) = V_k,\ \mathscr{K}(k) = \mathscr{C}_k \\ \quad\quad k = k+1 \\ \mathscr{C} = (\mathscr{V}, \mathscr{K}, x) \end{cases} \tag{12}$$

[18] Kai Weng Wong, Rüdiger Ehlers & Hadas Kress-Gazit (2014): *Correct High-level Robot Behavior in Environments with Unexpected Events.* In: *Robotics: Science and Systems.*

A Fixed-point Operators

In this appendix, we present algorithms from [12] to compute the winning set for a specification in the form of (1). In addition to the winning set, these algorithms provide an explicit construction of the control implementation as in Definition 5.

The most general algorithm that computes the winning set and the controller for specifications in the form of (1) is given in (9). The winning set that results from (9) is equal to V_∞, i.e. the limit of the expanding sequence V_k. $\mathscr{C}$ is the controller corresponding to the winning set V_∞.

The building blocks for algorithm (9) are (fixed-point based) operators (10), (11), (12), (13) and (14). Each operator corresponds to a type of LTL formula used in (9). Note that when a fixed-point operator appears as part of a formula, it only refers to its first output, i.e. the winning set.

The winning sets resulting from (10), (11), (14) are W_∞, X_∞ and Y_∞. In words, $\mathrm{Pre}^{T,U}_{\exists,\forall}$ is a one-step reachability operator, and $\mathrm{Inv}^{U,G}_{\exists}(Z,B)$ computes $Y_\infty \subseteq (G \cap B) - Z$ from where the state can be controlled (with actions in U) to either remain inside Y_∞ or reach Z, but because G is a progress group under U,

$$[W,\mathscr{C}] = \mathrm{Pre}^{T,U}_{\exists,\forall}(V) = \begin{cases} W = \{q_1 \in Q : \exists(u \in U)\forall(q_2 \\ \qquad s.t.\ (q_1,u,q_2) \in \rightarrow_T), q_2 \in V\} \\ D(q) = \{u \in U : \forall(q_2 \\ \qquad s.t.\ (q,u,q_2) \in \rightarrow_T),\ q_2 \in V\} \\ \mathscr{C} = \{(q,D(q)) : q \in Q, D(q) \neq \emptyset\} \end{cases} \tag{13}$$

$$[Y_\infty,\mathscr{C}] = \mathrm{Inv}^{D,G}_{\exists}(Z,B) = \begin{cases} Y_0 = (G \cap B) - Z \\ \mathit{while}\ Y_{k+1} \neq Y_k : \\ \quad Y_{k+1} = Y_k \cap \mathrm{Pre}^{T,D}_{\exists,\forall}(Y_k \cup Z) \\ [-,\mathscr{C}] = \mathrm{Pre}^{T,D}_{\exists,\forall}(Y_k \cup Z) \\ Y_\infty = Y_k,\ \mathscr{C}\ \mathit{restricted\ to}\ Y_\infty \end{cases} \tag{14}$$

remaining indefinitely in G is impossible and therefore Z is eventually reached, which is why Y_∞ is part of the winning set for $B\ \mathbf{U}\ Z$. On the other hand, $\mathrm{PGPre}^{T}_{\exists,\forall}(Z,B)$ calls $\mathrm{Inv}^{D,G}_{\exists}(Z',B)$ over all the progress groups, to collect the feasible winning states inside progress groups where $B\ \mathbf{U}\ Z$ can be enforced.

For controllers resulting from (9), (10), (11) and (12), their descendant controllers come from the outputs of operators they call internally. Leaf nodes in a controller always consist of simple controllers, i.e. the ones given by (13) and (14).

B Proof of Theorem 4

Proof: Assume that Y_t and $\mathscr{C}_t$ are the winning set and controller resulting from $\mathrm{Inv}^{D,G}_{\exists}(\widehat{Z})$. We want to show that $Y_t = \widehat{Y}$, which immediately implies $\mathscr{C}_t = \widehat{\mathscr{C}}$.

The winning set Y of $\mathrm{Inv}^{D,G}_{\exists}(Z)$ is the largest subset of $(G \cap B) - Z$ satisfying the convergence condition w.r.t Z, i.e. $Y \subseteq Pre^{T,D}_{\exists,\forall}(Y \cup Z)$ and Y is the greatest fixed-point of this operator, so is Y_t w.r.t. $\widehat{Z}$. Also, $Y_t \subseteq Y_0$ for Y_0 in (3) by Theorem 1.

First, we show $Y_t \subseteq \widehat{Y}$: Since $Y_t \subseteq Y_0$ and $Y_k = Y_0 - \bigcup_{i \leq k} \Delta Y_i$ for Y_0, ΔY_k and Y_k in (3), it is enough to show $\Delta Y_k \cap Y_t = \emptyset$, for all k. Proceeding by induction: $\Delta Y_0 \cap (Y_t \cup \widehat{Z}) = \emptyset$. Assume that $\Delta Y_k \cap (Y_t \cup \widehat{Z}) = \emptyset$, then it can be easily checked that $\mathrm{Pre}^{T_k,D}_{\forall,\exists}(\Delta Y_k) \cap \mathrm{Pre}^{T,D}_{\exists,\forall}(Y_t \cup \widehat{Z}) = \emptyset$ based on the fact that $\rightarrow_{T_k} \subseteq \rightarrow_T$. For $Y_t \subseteq \mathrm{Pre}^{T,D}_{\exists,\forall}(Y_t \cup \widehat{Z})$ and $\Delta Y_{k+1} = \Delta Y_k \cup \mathrm{Pre}^{T_k,D}_{\forall,\exists}(\Delta Y_k)$, $Y_t \cap \Delta Y_{k+1} = \emptyset$. Hence by induction argument, $Y_t \cap \Delta Y_k = \emptyset$ for all k, i.e. $Y_t \subseteq \widehat{Y}$.

Second, we show $\widehat{Y} \subseteq Y_t$: It is enough to show that $\widehat{Y}$ satisfies the convergence condition w.r.t. $\widehat{Z}$. By definition of T_k in (3), it is easy to check that $Y_k = \{s_1 : \exists u, \exists s_2\ s.t.(s_1,u,s_2) \in \rightarrow_{T_k}\}$. Therefore $\mathrm{Pre}^{T_k,D}_{\forall,\exists}(\Delta Y_k) \subseteq Y_k$ for all k by definition of $\mathrm{Pre}^{T_k,D}_{\forall,\exists}$ in (3). Then we can express $\Delta Y_{k+1} = \Delta Y_k \cup (Y_k \cap \mathrm{Pre}^{T,D}_{\forall,\exists}(\Delta Y_k))$. Based on the new expression of ΔY_{k+1} and $Y_k \cap \widehat{Z} = \emptyset$, we have $\Delta Y_k = Q - (Y_k \cup \widehat{Z})$. The limit of redefined ΔY_k exists, for it is increasing and contained by a finite set Q. Once ΔY_k converges, $Y_k \cap \mathrm{Pre}^{T_k,D}_{\forall,\exists}(\Delta Y_k) \subseteq \Delta Y_k$. Since $Y_k = \widehat{Y}$ and ΔY_k are disjoint, we have $\widehat{Y} \cap \mathrm{Pre}^{T_k,D}_{\forall,\exists}(\Delta Y_k) = \emptyset$, i.e. $\widehat{Y} \cap \mathrm{Pre}^{T_k,D}_{\forall,\exists}(Q - (\widehat{Y} \cup \widehat{Z})) = \emptyset$. This being empty set is equivalent to $\forall s_1 \in \widehat{Y}$, not $(\forall u \in D, \exists s_2 \in Q - (\widehat{Y} \cup \widehat{Z}), (s_1,u,s_2) \in \rightarrow_T)$, i.e. $\forall s_1 \in \widehat{Y}, \exists u \in D, \forall s_2 \in Q - (\widehat{Y} \cup \widehat{Z}), (s_1,u,s_2) \notin \rightarrow_{T_k}$. That implies that $\forall s_1 \in \widehat{Y}, \exists u \in D, \forall (s_2\ s.t.\ (s_1,u,s_2) \rightarrow_{T_k}), s_2 \in (\widehat{Y} \cup \widehat{Z})$. By definition of T_k in (3), for all $s_1 \in \widehat{Y}$ and $u \in D$, if $(s_1,u,s_2) \in \rightarrow_T$, $(s_1,u,s_2) \in \rightarrow_{T_k}$. So we have $\forall s_1 \in \widehat{Y}, \exists u \in D, \forall (s_2\ s.t.\ (s_1,u,s_2) \rightarrow_T), s_2 \in (\widehat{Y} \cup \widehat{Z})$, that is $\widehat{Y} \subseteq \mathrm{Pre}^{T,D}_{\exists,\forall}(\widehat{Y} \cup \widehat{Z})$. Thus, $\widehat{Y}$ satisfies the convergence condition over $\widehat{Z}$. ■