Warm-Starting Fixed-Point Based Control Synthesis

Anonymous Submission

ABSTRACT

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 hopping robot 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 an order of magnitude speed-up compared to synthesizing a controller from scratch.

CCS CONCEPTS

• **Computer systems organization → Robotic control**;

KEYWORDS

control synthesis, formal method, warm-start, fixed-point

ACM Reference Format:

Anonymous Submission. 2018. Warm-Starting Fixed-Point Based Control Synthesis. In *Proceedings of International Conference on Embedded Software (EMSOFT'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Control synthesis techniques for discrete systems have been a central topic both for reactive synthesis and discrete-event systems [9, 10] with recent results establishing a connection between the two communities [4, 11]. 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 [9]. Therefore, some research has focused on identifying fragments of LTL that have favorable complexity (see e.g., [2, 3, 8, 13]). 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 [6, 7, 14]. Livingston et al. [7] 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 system and environment only break an existing controller locally, they re-synthesize a controller only for the affected nodes and replace the broken part with the new one. When "localness" assumption or estimation of the affected states fails, the performance of this method could degrade to the case that does synthesis from scratch. This method is also extended to handle changes in the specification such as addition of new goal regions [6]. Wong et al. [14] 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 hopping robot on a constrained surface, as described later. 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., [8, 13]). 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 modifies 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

EMSOFT'18, September 2018, Torino, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 hopping robot. 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 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-start 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 hopping 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 $A - B$. A function or a map $F : C \to D$ corresponds to the set $\{(c, F(c)) : c \in C, F(c) \in D\}$.A list $\mathcal{V} = [x_1, ..., x_n]$ is a totally ordered set, where $\mathcal{V}(i)$ refers to its ith-order element. To assign x_i to be the ith element in $\mathcal{V}$ is written as $\mathcal{V}(i) = x_i$. Not sure it is really useful to define functions or maps here, the terms are barely used in the rest of the document (function x4, map x1)

2.2 Augmented Finite Transition Systems

Definition 2.1. An **augmented finite transition system** (AFTS) [8] is a tuple $T = (Q, U, \to_T, G, AP, h_Q)$, where Q is a set of states, U is a set of actions (control inputs), $\to_T \in Q \times U \times Q$ is is a transition relation between states under specific actions, $G : 2^U \to 2^{2^Q}$ 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 $h_Q : Q \to 2^{AP}$ is a labeling function, which maps states to the set of atomic propositions they preserve verify.

Definition 2.2. A **control strategy** for an AFTS is a partial function $\mu : (Q \times U)^* \times Q \to 2^U$ which maps the history of state-action pairs and the current state to a set of actions.

[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. (see [5, 8, 12])

2.3 Linear Temporal Logic

Linear Temporal Logic (LTL) can be utilized to describe the desired behaviors of an AFTS system. It has logic operators (negation ¬, conjunction ∧, disjunction ∨), and temporal operators (next ◯, always □, eventually ◇ and until **U**).

use grammar to define LTL, cite Baier and Katoen

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

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

what is the numbering vi, v, etc. below? where $p \in AP$, ϕ_1 and ϕ_2 are also LTL formulas. The other operators can be derived as follows: $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$, $\phi_1 \implies \phi_2 = \neg\phi_1 \vee \phi_2$, $\Diamond\phi = True\ \mathbf{U}\ \phi$, $\Box\phi = \neg\Diamond\neg\phi$.

Semantics of LTL [8]: An ω-word is an infinite sequence in 2^{AP}. The satisfaction of an LTL specification ϕ 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 of an AFTS $y = \{y(n)\}_{n=0}^{\infty}$, the *$\omega$-words* of y is $h_Q(y) = \{h_Q(y(n))\}_{n=0}^{\infty}$, where h_Q is the labeling function of the AFTS. Given a LTL specification ϕ, we say that $y \models \phi$ if and only if $h_Q(y) \models \phi$.

2.4 Control Synthesis and Fixed-Point Operators

I just noticed G is used for progress group. Maybe we can use R (for recurrence) if it is not used for something else

The 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 subsets $A, B, R^i \subseteq Q$ defining atomic propositions. Such specifications can express properties of *invariance* ($\Box A$), *persistence* ($\Diamond\Box B$) and *reccurence*

Maybe we should say a few words on how this fragment differs from GR(1) and others

Definition 2.3. The **winning set** of the 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, specification can be enforced.

Definition 2.4. A **simple controller** for a winning set W over an AFTS is a function $C : \mathrm{W} \to 2^U$, i.e. a memoryless controller. A simple controller maps the current state to a set of actions under which the specification in the winning set remains guaranteed.

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

Definition 2.5. For the AFTS $T = \{Q, U, \rightarrow_T, G, AP, h_Q\}$, a **controller** C for a winning set $\mathrm{Win}^T(\phi)$ is a tuple $(\mathcal{V}, \mathcal{K}, x)$, where $\mathcal{V}$ is a list of subsets of Q, $\mathcal{K}$ is a list of controllers or simple controllers, i.e. so-called sub-controllers of C, and x is an internal variable that saves the index of sub-controllers called last time.

Overall, a controller is a tree structure with controllers at each node and simple controllers at the leaf nodes. The list $\mathcal{K}$ for each controller (node) denotes the children of that controller in the tree.

Winning sets and controllers for specifications in the form of (1) are computed iteratively via the following fixed-point based algorithm (see [8]).

$$[V_\infty, C] = \mathrm{Win}^{T,U}_{\exists,\forall}\left(\Box A \wedge \Diamond\Box B \wedge \left(\bigwedge_{i\in I} \Box\Diamond R^i\right)\right) \tag{2}$$

$$= \begin{cases} V_{\text{inv}} = \mathrm{Win}^{T,U}_{\exists,\forall}((A \mathbf{U} \emptyset) \vee \Box(A \wedge \Diamond Q)) \\ \text{Restrict synthesis to } V_{\text{inv}} \\ V_0 = \emptyset,\ \mathcal{V} = \{\},\ \mathcal{K} = \{\} \\ \textit{while } V_{k+1} \neq V_k: \\ \quad Z_{k+1} = \mathrm{Pre}^{T,U}_{\exists,\forall}(V_k) \bigcup \mathrm{PGPre}^{T,U}_{\exists,\forall}(V_k, Q) \\ \quad [V_{k+1}, C_{k+1}] = \mathrm{Win}^{T}_{\exists,\forall}((B \mathbf{U} Z_{k+1}) \vee \Box(B \wedge (\wedge_{i\in I} \Diamond R^i)) \\ \quad \mathcal{V}(k+1) = V_{k+1},\ \mathcal{K}(k+1) = C_{k+1} \\ V_\infty = V_k,\ C = (\mathcal{V}, \mathcal{K}, x) \end{cases} \tag{3}$$

The winning set that results from (3) is equal to V_∞, i.e. the limit of the expanding sequence V_k. C is the controller corresponding to the winning set V_∞.

The building blocks of the above algorithm are the following fixed-point operators. Each operator corresponds to a type of LTL formula used in (3). (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.)

$$[W_\infty, C] = \mathrm{Win}^{T,U}_{\exists,\forall}((B \mathbf{U} Z) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))) \tag{4}$$

$$= \begin{cases} W_0 = Q,\ \mathcal{V} = \{\},\ \mathcal{K} = \{\} \\ \textit{while } W_{k+1} \neq W_k: \\ \quad Z^i_{k+1} = Z \cup (B \cap R^i \cap \mathrm{Pre}^{T,U}_{\exists,\forall}(W_k)) \\ \quad [X^i, C^i] = \mathrm{Win}^{T,U}_{\exists,\forall}(B \mathbf{U} Z^i_{k+1}), \forall i \in I \\ \quad W_{k+1} = \bigcap_{i\in I} X^i \\ \mathcal{V}(1) = W_k,\ \mathcal{V}(i+1) = B \cap R^i,\ \mathcal{K}(i) = C^i, \forall i \in I \\ W_\infty = W_k,\ C = (\mathcal{V}, \mathcal{K}, x) \end{cases} \tag{5}$$

$$[X_\infty, C] = \mathrm{Win}^{T,U}_{\exists,\forall}(B \mathbf{U} Z) \tag{6}$$

$$= \begin{cases} X_0 = \emptyset,\ \mathcal{V} = \{\},\ \mathcal{K} = \{\} \\ \textit{while } X_{k+1} \neq X_k \\ \quad [V^1_k, C^1_k] = \mathrm{Pre}^{T,U}_{\exists,\forall}(X_k) \\ \quad [V^2_k, C^2_k] = \mathrm{PGPre}^{T,U}_{\exists,\forall}(Z \cup (B \cap V^1_k), B) \\ \quad X_{k+1} = Z \cup (B \cap V^1_k) \cup V^2_k \\ \quad \mathcal{V}(2k-1) = V^1_k, \mathcal{V}(2k) = V^2_k \\ \quad \mathcal{K}(2k-1) = C^1_k,\ \mathcal{K}(2k) = C^2_k \\ X_\infty = X_k,\ C = (\mathcal{V}, \mathcal{K}, x) \end{cases} \tag{7}$$

$$[Z_\infty, C] = \mathrm{PGPre}^{T,U}_{\exists,\forall}(Z, B) \tag{8}$$

$$= \begin{cases} Z_\infty = Z,\ \mathcal{V} = \{\},\ \mathcal{K} = \{\}, k = 1 \\ \textit{for } U \in 2^U: \\ \quad \textit{for } G \in G(U): \\ \quad\quad [V_k, C_k] = Inv^{U,G}_{\exists}(Z_\infty, B) \\ \quad\quad Z_\infty = Z_\infty \cup V_k \\ \quad\quad \mathcal{V}(k) = V_k,\ \mathcal{K}(k) = C_k,\ k++ \\ C = (\mathcal{V}, \mathcal{K}, x) \end{cases} \tag{9}$$

$$[W, C] = \mathrm{Pre}^{T,U}_{\exists,\forall}(V) \tag{10}$$

$$= \begin{cases} W = \{q_1 \in Q : \exists(u \in U)\forall(q_2\ s.t.\ (q_1, u, q_2) \in \rightarrow_T), q_2 \in V\} \\ D(q) = \{u \in U : \forall(q_2\ s.t.\ (q, u, q_2)),\ q_2 \in V\} \\ C = \{(q, D(q)) : q \in Q, D(q) \neq \emptyset\} \end{cases} \tag{11}$$

$$[Y_\infty, C] = \mathrm{Inv}^{D,G}_{\exists}(Z, B) \tag{12}$$

$$= \begin{cases} Y_0 = (G \cap B) - Z \\ \textit{while } Y_{k+1} \neq Y_k: \\ \quad Y_{k+1} = Y_k \cap \mathrm{Pre}^{T,D}_{\exists,\forall}(Y_k \cup Z) \\ [-, C] = Pre^{T,D}_{\exists,\forall}(Y_k \cup Z) \\ Y_\infty = Y_k,\ C\ \textit{restricted to}\ Y_\infty \end{cases} \tag{13}$$

The winning sets resulting from (5), (7), (13) are W_∞, X_∞ and Y_∞. $\mathrm{Pre}^{T,U}_{\exists,\forall}$ 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, remaining indefinitely in G is impossible and therefore Z is eventually reached, which is why Y_∞ is a winning set for $B \mathbf{U} Z$.

Definition 2.6. **Execution of controllers**: In the execution time, a controller $C = (\mathcal{V}, \mathcal{K}, x)$ acts as a function that maps the current state in Q to a set of feasible actions in U, for the AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$. Given the current state s, the output $C(s)$ is determined in two steps:

(i) update the internal variable x of C:
- if C is returned by (3),(7),(13):

$$x^+ = \begin{cases} \arg\min_{y \leq x}\{s \in \mathcal{V}(y)\}, & if\ x \neq 0 \\ \arg\min_y\{s \in \mathcal{V}(y)\}, & otherwise \end{cases}$$

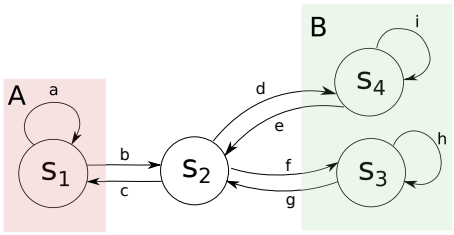

Figure 1: A simple finite transition system with state space $\{s1, s2, s3, s4\}$ and action space $\{a-h\}$, used to illustrate how the controller works. Atomic propositions are $A = \{s_1\}$ and $B = \{s_3, s_4\}$.

– if C is returned by (5):

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

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

(ii) Return the output of the x^+th sub-controller in $\mathcal{K}$ with input s, i.e.

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

If $\mathcal{K}$ is a list of simple controllers, $\mathcal{K}(x^+)(s)$ returns the set of feasible actions and this function call ends, otherwise above process repeats recursively until a simple controller is reached.

For me there is a problem of clarity with the definition of controllers, especially with the internal variable. I think we should explain in the definition roughly what happens when a call to the controller is made: if I understood well the memory variable changes according to some finite state automaton (we could say that and say that an example of how it works precisely is given below. But if the controller indeed needs the transition graph it is not just a tuple, we should add the transitions of the internal variables in the definition. Or am I missing something? Also overall the way this transition graph is constructed from the fixed-point operators (at least an intuition of it) is missing.

Example 2.7. For a simple transition system shown in Figure 1, given the specification $\phi = \square\diamond A \wedge \square\diamond B$, the outputs of (3) are winning set $W = \{s_1, s_2, s_3, s_4\}$, and controller $C = (W, \{C_0^0\}, x = 0)$, where the sub-controllers are:

$C_0^0 = (\{W, \{s_1\}, \{s_3, s_4\}\}, \{C_0^1, C_1^1\}, x^0)$;
$C_0^1 = (\{\{s_1, s_2\}, W, W\}, \{C_i^{2,0}\}_{i=0}^2, x_0^1)$;
$C_1^1 = (\{\{s_2, s_3, s_4\}, W, W\}\{C_i^{2,1}\}_{i=0}^2, x_1^1), C_0^{2,0} = \{(s_1, \{a\}), (s_2, \{c\})\}$;
$C_1^{2,0} = \{(s_1, \{a, b\}), (s_2, \{c\}), (s_3, \{g\}), (s_4, \{e\})\}$;
$C_2^{2,0} = \{(s_1, \{a, b\}), (s_2, \{c, d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$;
$C_0^{2,1} = \{(s_2, \{d, f\}), (s_3, \{h\}), (s_4, i)\}$;
$C_1^{2,1} = \{(s_1, \{b\}), (s_2, \{d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$;
$C_2^{2,1} = \{(s_1, \{a, b\}), (s_2, \{c, d, f\}), (s_3, \{g, h\}), (s_4, \{e, i\})\}$.

C, C_0^0 and C_i^1 are returned by (3), (5) and (7), with internal variables x^j or x_i^j. $C_i^{2,k}$ are simple controllers returned by (11). At each time interval, (x, x^0, x_0^1, x_1^1) will be updated according to the update rules in Figure 2, and then controllers $C_x^0, C_{x^0}^1$ and $C_{x_k^1}^{2,k}$ (if $x^0 = k$) will be called to determine the control input.

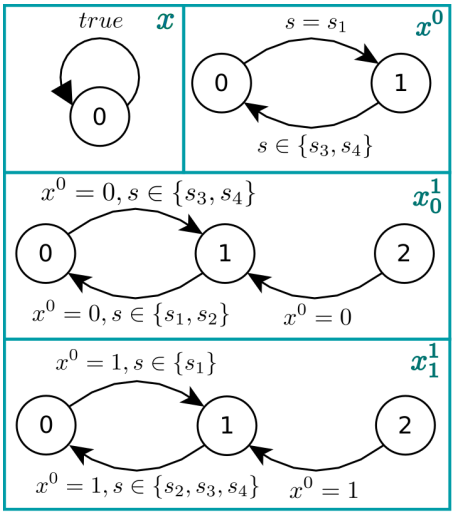

Figure 2: Update x first, and then x^0, and lastly x_0^1 and x_1^1. s is the current state. The transition condition is labeled on the edge. Self loop exists (ignored on the graph): one variable keeps unchanged if no condition satisfies.

For example, take the initial value of internal variables as $(0, 0, 0, 0)$. For the initial state s_1, updating internal variables following Figure 2 gives $(0, 1, 0, 1)$. The set of feasible control inputs is $\{b\} = C_1^{2,1}(s_1)$. Following this procedure and always taking the first element in the set feasible inputs, the trajectory under control of 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.

don't use b.w., s,t, use between, such that. in general, it is better writing practice to avoid abbreviations as much as possible

2.5 Problem Statement

In this section, we formally define the problem of interest. it is usually a good idea to start with telling what you want to do in the section so that the user is prepared.

Somewhere we should give the main steps of the solution also. I think one thing that would be important is to state clearly what can and cannot change in the structure of the controller when U is modified. I guess it is not only a change of the transitions, like in the example, since in some cases the number of steps to convergence can be modified. But then what exactly remains invariant? Only the "depth" of the controller?

Problem 1. *Given an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$, a LTL specification $\phi = \square A \wedge \diamond\square B \wedge \left(\bigvee_{i\in I} \square\diamond R^i\right)$, the winning set W_ϕ and controller C_ϕ, if a set of actions $U_d \subseteq U$ in T is unavailable, find the new winning set and a controller based on W_ϕ and C_ϕ.*

3 PATCHING METHOD

Before presenting the patching methods, we state a property of controller implementations that roughly shows that the controllers

contain all the relevant information needed for patching. We start with a definition of strategy containment.

Definition 3.1. For two control strategies μ_1 and μ_2 with winning set W_1 and W_2, we say μ_2 contains μ_1, denoted as $\mu_1 \subseteq \mu_2$, if $\mu_1(\{(s_i, a_i)\}_{i=1}^{n-1}, s_n)$ is a subset of $\mu_2(\{(s_i, a_i)\}_{i=1}^{n-1}, s_n)$, for all feasible n and all transition trajectories $q = (s_1, a_1), (s_2, a_2), ..., (s_{n-1}, a_{n-1})$ satisfying $s_1 \in W_1$, $a_k \in \mu_1(\{(s_i, a_i)\}_{i=1}^{k-1}, s_k)$.

Consider an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$ and an LTL specification $\phi = \Box A \wedge \Diamond\Box B \wedge \left(\bigvee_{i\in I} \Box\Diamond G^i\right)$. Compute its winning set (3) and the winning control strategy μ_ϕ (or controller C_ϕ) via fixed-point operators. Now disable some actions $U_d \subseteq U$, i.e. getting a new AFTS $\widehat{T} = (Q, U - U_d, \rightarrow_T, G, AP, h_Q)$ and re-compute the winning set and control strategy, denoted by $\widehat{W}_\phi$ and $\widehat{\mu}_\phi$ (or controller $\widehat{C}_\phi$), via the same fixed-point operator. Then, we have the following.

Theorem 3.2. *For $W_\phi, \mu_\phi, \widehat{W}_\phi, \widehat{\mu}_\phi$ defined above, we have $\widehat{W}_\phi \subseteq W_\phi$ and $\widehat{\mu}_\phi \subseteq \mu_\phi$.*

Proof. (i) For any state $s \in \widehat{W}_\phi$, if the system $\widehat{T}$ starting from s could be controlled under an infinite sequence of actions and satisfy the specification, so is T, since T can apply exactly the same sequence of actions and get the same trajectory. Hence $s \in W_\phi$, for (3) is sound and complete (See Lemma 13 in [8]). Therefore $\widehat{W}_\phi \subseteq W_\phi$. (ii) any control strategy in $\widehat{T}$ works in T by construction, which implies $\widehat{\mu}_\phi \subseteq \mu_\phi$. □

If C_ϕ is implemented to contain all the possible control strategies, Theorem 3.2 says that C_ϕ should contain all the information of $\widehat{C}_\phi$, which implies there is a possibility to obtain $\widehat{C}_\phi$ by modifying the existing controller.

3.1 Patching Simple Controllers

Definition 3.3. Given a simple controller C over an AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$, a **finite transition system** (FTS) corresponding to C is $T(C) = (Q_C, U_C, \rightarrow_{T(C)})$, where Q_C is the set of states that appear in $\rightarrow_{T(C)}$, $U_C = \{u : u \in C(s), \forall s \in W\}$, $\rightarrow_{T(C)} = \{(s_1, u, s_2) \in \rightarrow_T : s_1 \in W, s_2 \in Q, u \in C(s_1)\}$.

Remark 1. *By Definition 3.3, $T(C)$ can be easily constructed from the simple controller C and the AFTS T. Conversely, C can be constructed from $T(C)$ by $C = \{(s, D(s)) : s \in W, D(s) \neq \emptyset\}$, where $D(s) = \{u : (s, u, s_2) \in \rightarrow_{T(C)}\}$.*

Due to the easy conversion between C and $T(C)$, we can modify a simple controller C by modifying its corresponding FTS $T(C)$ and then converting $T(C)$ to C, with the advantages that $T(C)$ contains the necessary transition information, which C doesn't have, and is easier to manipulate.

Assume the AFTS $T = (Q, U, \rightarrow_T, G, AP, h_Q)$ is the abstraction of the system that needs to be controlled. Assume that $T(C) = (Q_C, U_C, \rightarrow_{T(C)})$ is a transition system corresponding to a simple controller C.

Firstly let's define several operation to manipulate the finite transition system $T(C)$:

transitions removing: given $E = \{(s_1, a, s_2) : s_1, s_2 \in Q_C, a \in U_C\}$, remove all transitions in E from $T(C)$ i.e. replace $\rightarrow_{T(C)}$ with $\rightarrow_{T(C)} - E$, written as

$$T(C)\backslash E$$

There are three variations of this operation:

- $T(C)\backslash(E, *) \equiv T(C)\backslash\{(s_1, a, s_2) \in \rightarrow_{T(C)} : (s_1, a) \in E)\}$, where $E \subseteq Q_C \times U_C$.
- $T(C)\backslash(*, U_d, *) \equiv T(C)\backslash\{(s_1, a, s_2) \in \rightarrow_{T(C)} : a \in U_d\}$, where $U_d \subseteq U_C$.
- $T(C)\backslash(Q_d, *, *) \equiv T(C)\backslash\{(s_1, a, s_2) \in \rightarrow_{T(C)} : s_1 \in Q_d\}$, where $Q_d \subseteq Q_C$

transition adding: given $E \subseteq Q \times U \times Q$, replace $\rightarrow_{T(C)}$ with $\rightarrow_{T(C)} \cup E$, denoted for simplicity by

$$T(C) \cup E$$

Note that since $E \subseteq Q \times U \times Q$, E may contain states and actions which do not exists in $Q_C \subseteq Q$ and $U_C \subseteq U$. Then just assume that Q_C and U_C will enlarge automatically, which is easy to implement in code.

null node seeking: return the set $\{s_1 \in Q_{T(C)} : \exists(s_1, a, s_2) \in \rightarrow_{T(C)}, \forall a \in U_{T(C)}, \forall s_2 \in Q_{T(C)}\}$, i.e. all the nodes whose out-degree is zero, written as

$$Vac(T(C))$$

state-action pre: return the set $\{(s_1, a) : (s_1, a, s_2) \in \rightarrow_{T(C)}, s_2 \in S_2\}$, i.e. all the edges pointing into nodes in S_2, written as

$$\widehat{Pre}_{\exists,\exists}^{T(C),U}(S_2)$$

Now we are ready to modify simple controllers returned by (11) and (13):

Assume that W and C are the winning set and controller returned by $\text{Pre}_{\exists,\forall}^{T,U}(V)$.

Given $\widehat{V} = V - \Delta V$ and $\widehat{U} = U - U_d$, we want to modify W and C to be the winning set and controller of $\text{Pre}_{\exists,\forall}^{T,\widehat{U}}(\widehat{V})$. The patching operator is:

$$[\widehat{W}, \widehat{C}] = \overline{\text{Pre}}_{\exists,\forall}^{T,U}(C, \Delta V, U_d) \tag{14}$$

$$= \begin{cases} T_0 = T(C)\backslash(*, U_d, *) \\ E = \widehat{Pre}_{\exists,\exists}^{T_0, U-U_d}(\Delta V) \\ T(\widehat{C}) = T_0\backslash E \\ \widehat{W} = \{s_1 : \exists u, \exists s_2 s.t.(s_1, u, s_2) \in \rightarrow_{T(\widehat{C})}\} \end{cases} \tag{15}$$

where $\widehat{C}$ is converted from $T(\widehat{C})$. $\widehat{W}$ and $\widehat{C}$ will be equal to the outputs of $\text{Pre}_{\exists,\forall}^{T,\widehat{U}}(\widehat{V})$.

Proof. Let's assume that $\text{Pre}_{\exists,\forall}^{T,\widehat{U}}(\widehat{V})$ returns W_t and C_t. It's enough to show that $T(\widehat{C}) = T(C_t)$, i.e. $\rightarrow_{T(\widehat{C})} = \rightarrow_{T(C_t)}$.

By (11) and Definition 3.3, it's obvious that $\rightarrow_{T(C_t)} \subseteq \rightarrow_{T(C)}$ for $\widehat{V} \subseteq V$ and $\widehat{U} \subseteq U$. The set of transitions $S = (*, U_d, *) \cup E$ is only related to states in ΔV or actions in U_d, so $\rightarrow_{T(C_t)}$ and S are disjoint. $\rightarrow_{T(\widehat{C})} = \rightarrow_{T(C)} - S$. Thus $\rightarrow_{T(C_t)} \subseteq \rightarrow_{T(\widehat{C})}$

By (15), transitions in $T(\widehat{C})$ can only take actions in $\widehat{U}$ and transit to states in $\widehat{V}$, which implies that $\rightarrow_{T(\widehat{C})} \subseteq \rightarrow_{T(C_t)}$. □

Next, Y and C assume that the winning set and controller resulting from $\text{Inv}_{\exists}^{D,G}(Z,B)$. Assume that the set of unavailable actions is U_d. If $D \cap U_d \neq \emptyset$, the winning set would be empty by definition of progress group (see[8]). Given that $\widehat{Z} \subseteq Z$ and $D \cap U_d = \emptyset$, we want to modify W and C to be the winning set and controller for $\text{Inv}_{\exists}^{D,G}(\widehat{Z},B)$. The patching operator is

$$[\widehat{Y},\widehat{C}] = \overline{Inv}_{\exists}^{D,G}(Y,C,Z,\widehat{Z}) \tag{16}$$

$$= \begin{cases} Y_0 = Y \cup (\Delta Z \cap G \cap B) \\ \Delta Y_0 = Q - (Y_0 \cup \widehat{Z}) \\ T_0 = T(C) \cup E_0 \\ \textit{while } Y_{k+1} \neq Y_k : \\ \quad \Delta Y_{k+1} = \text{Pre}_{\forall,\exists}^{T_k,D}(\Delta Y_k) \\ \quad T_{k+1} = T_k \backslash (\Delta Y_{k+1},*,*) \\ \quad Y_{k+1} = Y_k - \Delta Y_{k+1} \\ \widehat{Y} = Y_k,\ T(\widehat{C}) = T_k \end{cases} \tag{17}$$

where $\Delta Z = Z - \widehat{Z}$, and $E_0 = \{(s_1,a,s_2) \in \rightarrow_T : s_1 \in Y_0, a \in D, s_2 \in Q\}$. $\widehat{Y}$ is the modified winning set, and controller $\widehat{C}$ can be recovered from $T(\widehat{C})$, which will be the same as outputs from $\text{Inv}_{\exists}^{D,G}(\widehat{Z},B)$.

Requires careful reading

Proof. Assume that Y_t and C_t are the winning set and controller resulting from $\text{Inv}_{\exists}^{D,G}(\widehat{Z})$. Here we're going to prove that $Y_t = \widehat{Y}$. Once we have $Y_t = \widehat{Y}$, it's easy to check that $T(C_t) = T(\widehat{C})$.

By (13), the winning set Y of $\text{Inv}_{\exists}^{D,G}(Z)$ is the largest subset of $(G \cap B) - Z$ satisfying the convergence condition w.r.t Z, i.e. $Y \subseteq Pre_{\exists,\forall}^{T,D}(Y \cup Z)$, so is Y_t w.r.t. $\widehat{Z}$.

First, prove that $Y_t \subseteq Y_0$ for Y_0 in (17): Check that $V = Y_t \cap (G \cap B - Z)$ is a subset of $G \cap B - Z$ satisfying the convergence condition over Z. For $V_1, V_2 \subseteq G \cap B - Z$ satisfying convergence condition, $V_1 \cup V_2$ satisfies convergence condition too, which implies $V \subseteq Y$ (otherwise, $Y \subseteq Y \cup V$ is a larger subset of $G \cap B - Z$ satisfying the convergence condition than Y. Contradiction!) So $Y_t = V \cup (Y_t \cap \Delta Z) \subseteq Y \cup (\Delta Z \cap G \cap B) = Y_0$.

Second, prove $Y_t \subseteq \widehat{Y}$: Since $Y_t \subseteq Y_0$ and $Y_k = Y_0 - \bigcup_{i \leq k} \Delta Y_i$, it's enough to show $\Delta Y_k \cap Y_t = \emptyset, \forall k$. Prove 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 $\text{Pre}_{\forall,\exists}^{T_k,D}(\Delta Y_k) \cap \text{Pre}_{\exists,\forall}^{T,D}(Y_t \cup \widehat{Z}) = \emptyset$ by (11) and the fact that $\rightarrow_{T_k} \subseteq \rightarrow_T$. For $Y_t \subseteq \text{Pre}_{\exists,\forall}^{T,D}(Y_t \cup \widehat{Z})$ and $\Delta Y_{k+1} = \text{Pre}_{\forall,\exists}^{T_k,D}(\Delta Y_k)$, $Y_t \cap \Delta Y_{k+1} = \emptyset$. Hence by induction argument, $Y_t \cap \Delta Y_k = \emptyset, \forall k$, i.e. $Y_t \subseteq \widehat{Y}$.

Finally, prove $\widehat{Y} \subseteq Y_t$: It's enough to show that $\widehat{Y}$ satisfies the convergence condition over $\widehat{Z}$. By definitions of T_k, $\Delta Y_{k+1} = Y_k \cap \text{Pre}_{\forall,\exists}^{T,D}(\Delta Y_k)$. Then by the additivity of $\text{Pre}_{\forall,\exists}^{T,D}$ (i.e. for any X_1 and X_2, $\text{Pre}_{\forall,\exists}^{T,D}(X_1 \cup X_2) = \text{Pre}_{\forall,\exists}^{T,D}(X_1) \cup \text{Pre}_{\forall,\exists}^{T,D}(X_2)$), it's easy to check that redefining fifth line in (17) by $\Delta Y_{k+1} = \Delta Y_k \cup (Y_k \cap \text{Pre}_{\forall,\exists}^{T,D}(\Delta Y_k))$ doesn't change Y_{k+1}. By the new definition of ΔY_k 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, $Y_k \cap \text{Pre}_{\forall,\exists}^{T,D}(\Delta Y_k) \subseteq \Delta Y_k$. Since $Y_k = \widehat{Y}$ and ΔY_k are disjoint, $\widehat{Y} \cap \text{Pre}_{\forall,\exists}^{T,D}(\Delta Y_k) = \emptyset$, i.e. $\widehat{Y} \cap \text{Pre}_{\forall,\exists}^{T,D}(Q - (\widehat{Y} \cup \widehat{Z})) = \emptyset$. It says that $\forall s_1 \in \widehat{Y}$, not $\forall u \in D, \exists s_2 \in Q, (s1,u,s2) \in \rightarrow_T$ and $s_2 \in (Q - (\widehat{Y} \cup \widehat{Z}))$, i.e. $\forall s_1 \in \widehat{Y}, \exists u \in D, \forall s_2 \in Q, (s_1,u,s_2) \notin \rightarrow_T$ or $s_2 \in (\widehat{Y} \cup \widehat{Z})$. That implies that $\widehat{Y} \subseteq \text{Pre}_{\exists,\forall}^{T,D}(\widehat{Y} \cup \widehat{Z})$, i.e. $\widehat{Y}$ satisfies the convergence condition over $\widehat{Z}$. □

Given a simple controller C and new synthesis settings, both (15) and (17) try to find $T(\widehat{C})$ inside $T(C)$. However, if we re-synthesize the new simple controller from scratch, (11) and (13) would try to find $T(C)$ in the whole AFTS T, which roughly needs more computation cost.

3.2 Patching Controllers

Based on the patching operators defined in the previous section, we start to modify the winning sets and controllers resulting from (9),(7),(5) and ultimately (3).

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

$$[\widehat{Z}_\infty,\widehat{C}] = \overline{\text{PGPre}}_{\exists,\forall}^{T,U}(C,Z,\widehat{Z},U_d) \tag{18}$$

$$= \begin{cases} Z_\infty = Z,\ \widehat{Z}_\infty = \widehat{Z},\ \widehat{\mathcal{V}} = \{\},\ \widehat{\mathcal{K}} = \{\},\ k = 1 \\ \textit{for } U \in 2^U : \\ \quad \textit{for } G \in G(U) : \\ \quad\quad \textit{if } U_d \cap U = \emptyset : \\ \quad\quad\quad [\widehat{\mathcal{V}}(k),\widehat{\mathcal{K}}(k)] = \overline{Inv}_{\exists}^{U,G}(\mathcal{V}(k),\mathcal{K}(k),Z_\infty,\widehat{Z}_\infty) \\ \quad\quad \textit{else} : \ \widehat{\mathcal{V}}(k) = \emptyset, \widehat{\mathcal{K}}(k) = \emptyset \\ \quad\quad Z_\infty = Z_\infty \cup \mathcal{V}(k),\ \widehat{Z}_\infty = \widehat{Z}_\infty \cup \widehat{\mathcal{V}}(k),\ k++ \\ \textit{Remove all } \emptyset \textit{ in } \widehat{\mathcal{V}} \textit{ and } \widehat{\mathcal{K}},\ \ \widehat{C} = (\widehat{\mathcal{V}},\widehat{\mathcal{K}},x) \end{cases} \tag{19}$$

Proof. As long as (17) works well, $\widehat{Z}_\infty$ and $\mathcal{V}(k)$ will be equal to those in $\text{PGPre}_{\exists,\forall}^{T,\widehat{U}}(\widehat{Z},B)$ for all k. The soundness and completeness of (17) have been proven, so (19) is sound and complete. □

Assume that X_∞ and $C = (\mathcal{V},\mathcal{K},x)$ are the winning set and controller resulting from $\text{Win}_{\exists,\forall}^{T,U}(B \,\mathbf{U}\, Z)$. Assume $|\mathcal{K}| = 2n$, i.e. (7) converges in n steps. Then, given $\widehat{Z} \subseteq Z$ and $\widehat{U} = U - U_d$, to get the winning set and controller for $\text{Win}_{\exists,\forall}^{T,\widehat{U}}(B \,\mathbf{U}\, \widehat{Z})$, the patching

operator is

$$[\widehat{X}_\infty, \widehat{C}] = \overline{\mathrm{Win}}^{T,U}_{\exists,\forall,(B\,\mathbf{U}\,Z)}(C, Z, \widehat{Z}, U_d) \tag{20}$$

$$= \begin{cases} X_0 = \emptyset, \widehat{X}_0 = \emptyset, \Delta X = \emptyset,\ \widehat{\mathcal{V}} = \{\},\ \widehat{\mathcal{K}} = \{\} \\ \textit{For } k \in \{0, 2, ..., n-1\} \textit{ and } \widehat{X}_{k+1} \neq \widehat{X}_k : \\ \quad [\widehat{\mathcal{V}}(2k+1), \widehat{\mathcal{K}}(2k+1)] = \overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathcal{K}(2k+1), X_k - \widehat{X}_k, U_d) \\ \quad E_k = Z \cup (B \cap \mathcal{V}(2k+1)),\ \widehat{E}_k = Z \cup (B \cap \widehat{\mathcal{V}}(2k+1)) \\ \quad [\widehat{\mathcal{V}}(2k+2), \widehat{\mathcal{K}}(2k+2)] = \overline{\mathrm{PGPre}}^{T,U}_{\exists,\forall}(\mathcal{K}(2k+2), E_k, \widehat{E}_k, U_d) \\ \quad X_{k+1} = Z \cup (B \cap \mathcal{V}(2k+1)) \cup \mathcal{V}(2k+2) \\ \quad \widehat{X}_{k+1} = \widehat{Z} \cup (B \cap \widehat{\mathcal{V}}(2k+1)) \cup \widehat{\mathcal{V}}(2k+2) \\ \textit{For } k \geq n \textit{ and } \widehat{X}_{k+1} \neq \widehat{X}_k : \\ \quad [\widehat{\mathcal{V}}(2k+1), \widehat{\mathcal{K}}(2k+1)] = \overline{\mathrm{Pre}}^{T,U}_{\exists,\forall}(\mathcal{K}(2n-1), X_n - \widehat{X}_k, U_d) \\ \quad \widehat{E}_k = Z \cup (B \cap \widehat{\mathcal{V}}(2k+1)) \\ \quad [\widehat{\mathcal{V}}(2k+2), \widehat{\mathcal{K}}(2k+2)] = \overline{\mathrm{PGPre}}^{T,U}_{\exists,\forall}(\mathcal{K}(2n), E_n, \widehat{E}_k, U_d) \\ \quad \widehat{X}_{k+1} = \widehat{Z} \cup (B \cap \widehat{\mathcal{V}}(2k-1)) \cup \widehat{\mathcal{V}}(2k) \\ \widehat{X}_\infty = \widehat{X}_k,\ \widehat{C} = (\widehat{\mathcal{V}}, \widehat{\mathcal{K}}, x) \end{cases} \tag{21}$$

The above patch algorithm has two parts: part one modifies the existing sub-controllers for $k \leq n$; part two adds new sub-controllers by modifying the nth existing sub-controller until convergence. The winning set $\widehat{X}_\infty$, and controller $\widehat{C}$ would be equal to the outputs of (7) with input parameters $\widehat{U}, B, \widehat{Z}$.

Proof. Let's denote X_k in (7) with inputs $\widehat{U}, B, \widehat{Z}$ as X^t_k. To show the soundness and completeness of (21), it's enough to show that $\widehat{X}_k$ in (21) is equal to X^t_k, for all k.

For $k < n$, $\widehat{X}_k = X^t_k$ is guaranteed by the correctness of (15) and (19).

For $k \geq n$, it's enough to show that $X^t_k \subseteq X_n$, then the correctness again is guaranteed by (15) and (19). According to (9), it's easy to show that $\mathrm{PGPre}^{T,U-U_d}_{\exists,\forall}(Z_1, B) \subseteq \mathrm{PGPre}^{T,U}_{\exists,\forall}(Z_1, B) \subseteq \mathrm{PGPre}^{T,U}_{\exists,\forall}(Z_2, B)$ for $Z_1 \subseteq Z_2$. Prove by induction: $X^t_0 = X_0$. Assume that $X^t_k \subseteq X_k$. Then $\mathrm{PGPre}^{T,U-U_d}_{\exists,\forall}(X^t_k, B) \subseteq \mathrm{PGPre}^{T,U-U_d}_{\exists,\forall}(X_k, B)$. It's trivial that $Pre^{T,U-U_d}_{\exists,\forall}(X^t_k, B) \subseteq Pre^{T,U-U_d}_{\exists,\forall}(X_k, B)$. Hence, $X^t_{k+1} \subseteq X_{k+1}$. By induction argument, $X^t_k \subseteq X_{k+1} \forall k$. But since $\forall k > n, X_k = X_n$. $X^t_k \subseteq X_n, \forall k > n$. □

Assume that W and $C = (\mathcal{V}, \mathcal{K}, x)$ are the winning set and controller resulting from $\mathrm{Win}^{T,U}_{\exists,\forall}((B\,\mathbf{U}\,Z) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))$. Given that $\widehat{Z} \subseteq Z$ and $\widehat{U} \subseteq U$, modify W and C for $\mathrm{Win}^{T,U}_{\exists,\forall}((B\,\mathbf{U}\,\widehat{Z}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i))$. The patching operator for (5), written as $\overline{\mathrm{Win}}^{T,U}_{\exists,\forall(\psi)}$ is

$$[\widehat{W}_\infty, \widehat{C}] = \overline{\mathrm{Win}}^{T,U}_{\exists,\forall(\psi)}(C, Z, \widehat{Z}, U_d) \tag{22}$$

$$= \begin{cases} W_0 = \mathcal{V}(1), \widehat{\mathcal{V}} = \mathcal{V}, \widehat{\mathcal{K}} = \mathcal{K} \\ Z^i_\infty = Z \cup (B \cap R^i \cap \mathrm{Pre}^{T,U}_{\exists,\forall}(W_0)) \\ \widehat{Z}^i_0 = \widehat{Z} \cup (B \cap R^i \cap \mathrm{Pre}^{T,U-U_d}_{\exists,\forall}(W_0)) \\ [X^i_0, \widehat{\mathcal{K}}(i)] = \overline{\mathrm{Win}}^{T,U}_{\exists,\forall,(B\mathbf{U}Z)}(\widehat{\mathcal{K}}(i), Z^i_\infty, \widehat{Z}^i_0, U_d) \\ \widehat{W}_0 = \bigcap_{i\in I} X^i_0 \\ \textit{while } \widehat{W}_{k+1} \neq \widehat{W}_k : \\ \quad \widehat{Z}^i_{k+1} = \widehat{Z} \cup (B \cap R^i \cap \mathrm{Pre}^{T,U-U_d}_{\exists,\forall}(\widehat{W}_k)) \\ \quad [X^i_{k+1}, \widehat{\mathcal{K}}(i)] = \overline{\mathrm{Win}}^{T,U}_{\exists,\forall,(B\mathbf{U}Z)}(\widehat{\mathcal{K}}(i), \widehat{Z}^i_k, \widehat{Z}^i_{k+1}, U_d) \\ \quad \widehat{W}_{k+1} = \bigcap_{i\in I} X^i_{k+1} \\ \widehat{W}_\infty = \widehat{W}_k,\ \widehat{C} = (\widehat{\mathcal{V}}, \widehat{\mathcal{K}}, x) \end{cases} \tag{23}$$

Proof. Assume that W_t is the winning set of (5) with input parameters $U - U_d$, B, C_i and $\widehat{Z}$. To show that $W_t = \widehat{W}_\infty$, it's enough to show that $W_t \subseteq \widehat{W}_\infty$, i.e. $\widehat{W}_\infty$ is a larger winning set than W_t. Then by the completeness of (5), $W_t = \widehat{W}_\infty$.

First of all, prove that $W_t \subseteq W_0$: For any trajectories satisfying $(B\,\mathbf{U}\,\widehat{Z}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i)$, it satisfies $(B\,\mathbf{U}\,Z) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i)$, so directly we have $W_t \subseteq W_0$.

Next, prove $W_t \subseteq \widehat{W}_k, \forall k$ by induction: Since $W_t \subseteq W_0$, $Z^i_t = \widehat{Z} \cup (B \cap R^i \cap \mathrm{Pre}^{T,U-U_d}_{\exists,\forall}(W_t)) \subseteq \widehat{Z}^i_0$. Hence $\mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,Z^i_t) \subseteq \mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_0)$ (proven by induction: show that X_k in (21) for Z^i_t is contained by X_k in (21) for $\widehat{Z}^i_0$, for all k). Then, $W_t = \bigcap_{i\in I} \mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,Z^i_t) \subseteq \bigcap_{i\in I} \mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_0) = \widehat{W}_0$, which is the base case. Assume that $W_t \subseteq \widehat{W}_k$. Then we have $Z^i_t \subseteq \widehat{Z}^i_{k+1}$ and $W_t \subseteq \widehat{W}_{k+1}$ for the same arguments in the base case. Hence by induction, $W^i_t \subseteq \widehat{W}_k, \forall k$.

Finally prove $\widehat{W}_k$ converges by induction: $\widehat{Z}^i_0 \subseteq Z^i_\infty$. Hence $\widehat{W}_0 = \bigcap_{i\in I} \mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_0) \subseteq \bigcap_{i\in I} \mathrm{Win}^{T,U}_{\exists,\forall}(B\,\mathbf{U}\,Z^i_\infty) = W_0$, which is the base case. Assume that $\widehat{W}_{k+1} \subseteq \widehat{W}_k$. Then $\widehat{Z}^i_{k+1} \subseteq \widehat{Z}^i_k$ and $\widehat{W}_{k+1} = \bigcap_{i\in I} \mathrm{Win}^{T,U-U_d}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_{k+1}) \subseteq \bigcap_{i\in I} \mathrm{Win}^{T,U}_{\exists,\forall}(B\,\mathbf{U}\,\widehat{Z}^i_k) = \widehat{W}_k$. By induction argument, $\{\widehat{W}_k\}$ is a decreasing sequence of finite sets, which converges surely. □

Now we are ready to modify controller in (3), since we have all the patching operators it requires.

Assume that W and $C = (\mathcal{V}, \mathcal{K}, x)$ are the winning set and controller resulting from

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

Assuming that $|\mathcal{K}| = n$. 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 (3), i.e. $[\mathcal{V}_1(k), \mathcal{K}_1(k)] = \mathrm{Pre}^{T,U}_{\exists,\forall}(V_k)$; the lists of winning sets and controllers returned by $\mathrm{PGPre}^{T}_{\exists,\forall}(V_k, B)$ in (3), i.e. $[\mathcal{V}_2(k), \mathcal{K}_2(k)] = \mathrm{PGPre}^{T}_{\exists,\forall}(V_k, B)$ $(k = 1, ..., n)$; the controller C_{Inv} returned by $\mathrm{Win}^{T,U}_{\exists,\forall}((A\mathbf{U}\emptyset) \vee \Box(A \wedge \Diamond Q))$ in the first line of (3). To restrict action space to $\widehat{U} = U - U_d$, the patching operator

for (3) is

$$[\widehat{V}_\infty, \widehat{C}] = \overline{\text{Win}}^{T,U}_{\exists,\forall,(\phi)}(C, C_{Inv}, \mathcal{V}_1, \mathcal{K}_1, \mathcal{V}_2, \mathcal{K}_2, U_d) \tag{24}$$

$$= \begin{cases} \widehat{V}_{\text{inv}} = \overline{\text{Win}}^{T,U}_{\exists,\forall,(\psi)}(C_{Inv}, \emptyset, \emptyset, U_d) \\ \textit{Restrict synthesis to } \widehat{V}_{\text{inv}} \\ \widehat{V}_0 = \emptyset,\ \mathcal{V}(0) = \emptyset,\ \widehat{\mathcal{V}} = \{\},\ \widehat{\mathcal{K}} = \{\} \\ \textit{for } k \in \{0, 1, 2, ..., n-1\} \textit{ and } \widehat{V}_{k+1} \neq \widehat{V}_k : \\ \quad \widehat{Z}^1_{k+1} = \overline{\text{Pre}}^{T,U}_{\exists,\forall}(\mathcal{K}_1(k+1), \mathcal{V}(k) - \widehat{V}_k, U_d) \\ \quad \widehat{Z}^2_{k+1} = \overline{\text{PGPre}}^{T,U}_{\exists,\forall}(\mathcal{K}_2(k+1), \mathcal{V}(k), \widehat{V}_k, U_d) \\ \quad Z_{k+1} = \mathcal{V}_1(k+1) \cup \mathcal{V}_2(k+1),\ \widehat{Z}_{k+1} = \widehat{Z}^1_{k+1} \bigcup \widehat{Z}^2_{k+1} \\ \quad [\widehat{V}_{k+1}, \widehat{\mathcal{K}}(k+1)] = \overline{\text{Win}}^{T,U}_{\exists,\forall,(\psi)}(\mathcal{K}(k+1), Z_{k+1}, \widehat{Z}_{k+1}, U_d) \\ \quad \widehat{\mathcal{V}}(k+1) = \widehat{V}_{k+1} \\ \textit{for } k \geq n \textit{ and } \widehat{V}_{k+1} \neq \widehat{V}_k : \\ \quad \widehat{Z}^1_{k+1} = \overline{\text{Pre}}^{T,U}_{\exists,\forall}(\mathcal{K}_1(n), \mathcal{V}(n-1) - \widehat{V}_k, U_d) \\ \quad \widehat{Z}^2_{k+1} = \overline{\text{PGPre}}^{T,U}_{\exists,\forall}(\mathcal{K}_2(n), \mathcal{V}(n-1), \widehat{V}_k, U_d) \\ \quad \widehat{Z}_{k+1} = \widehat{Z}^1_{k+1} \bigcup \widehat{Z}^2_{k+1} \\ \quad [\widehat{V}_{k+1}, \widehat{\mathcal{K}}(k+1)] = \overline{\text{Win}}^{T,U}_{\exists,\forall,(\psi)}(\mathcal{K}(n), Z_n, \widehat{Z}_{k+1}, U_d) \\ \quad \widehat{\mathcal{V}}(k+1) = \widehat{V}_{k+1} \\ \widehat{V}_\infty = \widehat{V}_k, \widehat{C} = (\widehat{\mathcal{V}}, \widehat{\mathcal{K}}, x) \end{cases} \tag{25}$$

The modified winning set $\widehat{V}_\infty$ and controller $\widehat{C}$ will be equal to outputs resulting from $\text{Win}^{T,U-U_d}_{\exists,\forall}\left(\Box A \wedge \Diamond\Box B \wedge \left(\bigwedge_{i\in I} \Box\Diamond R^i\right)\right)$. The same as (21), the patching algorithm has two parts: one for modifying existing sub-controllers; one for adding new ones until convergence.

Proof. The correctness of $\widehat{V}_{\text{inv}}$ is guaranteed by (23). For $k \leq n$, the correctness of $\widehat{V}_k$ is guaranteed by (15), (19) and (23). Assume that V^t_k and V_k are intermediate results in (3) with action space $U - U_d$ and U respectively. For $k > n$, the correctness is guaranteed by (15), (19) and (23) as long as $V^t_k \subseteq V_n$. We can prove it in the same way as the proof of (21). For simplicity, we can apply Theorem 1 instead: Assume that V^t_k converges to V^t_∞ and V_k converges to V_∞. By Theorem 3.2, we have $V^t_\infty \subseteq V_\infty = V_n$. Next, prove that $\{V^t_k\}$ is a monotonic increasing sequence of sets by induction. $\emptyset = V^t_0 \subseteq V^t_1$. Assume that $V^t_{k-1} \subseteq V^t_k$. Then it's not hard to see that $Z^t_k \subseteq Z^t_{k+1}$, and $V^t_k = \text{Win}^T_{\exists,\forall}((B \,\mathbf{U}\, Z_k) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i)) \subseteq \text{Win}^T_{\exists,\forall}((B \,\mathbf{U}\, Z_{k+1}) \vee \Box(B \wedge (\bigwedge_{i\in I} \Diamond R^i)) = V^t_{k+1}$ (this must be true otherwise (23) can't be correct). So $V^t_k \subseteq V_n, \forall k$.

□

4 EXAMPLE

4.1 Simple Transition System

For the transition system in Figure 1, take the controller we get in Section 2.4 and $U_d = \{e\}$ as input of (25), the outputs are the modified winning set $\widehat{W} = \{s_1, s_2, s_3\}$ and controller $\widehat{C} = (\{\widehat{W}\}, \{\widehat{C}^0_0\}, x = 0)$, where the sub-controllers are:

$\widehat{C}^0_0 = (\{\widehat{W}, \{s_1\}, \{s_2, s_3\}\}, \{\widehat{C}^1_0, \widehat{C}^1_1\}, x^0)$;
$\widehat{C}^1_0 = (\{\{s_1, s_2\}, W, W\}, \{\widehat{C}^{2,0}_i\}^2_{i=0}, x^1_0)$;
$\widehat{C}^1_1 = (\{\{s_2, s_3\}, W, W\}, \{\widehat{C}^{2,1}_i\}^2_{i=0}, x^1_1)$;

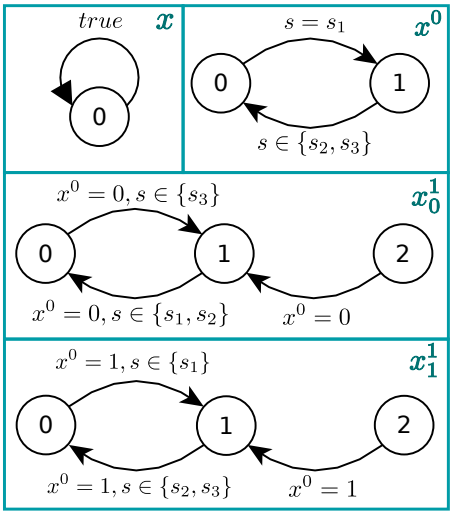

Figure 3: Transition graph of internal variables. Compared to the graph in Figure 1, transition conditions on most edges are different. What is the difference between the update rule and the transition conditions? I think there is space to put both transition graphs (before and after) on this figure. It would make it easier to see what changed.

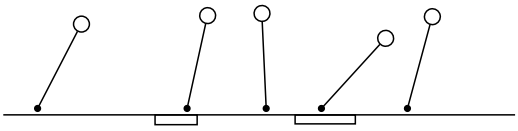

Figure 4: Hopping robots with point foot on a ground with holes.

$\widehat{C}^{2,0}_0 = \{(s_1, \{a\}), (s_2, \{c\})\}$;
$\widehat{C}^{2,0}_1 = \{(s_1, \{a, b\}), (s_2, \{c\}), (s_3, \{g\})\}$;
$\widehat{C}^{2,0}_2 = \{(s_1, \{a, b\}), (s_2, \{c, f\}), (s_3, \{g, h\})\}$;
$\widehat{C}^{2,1}_0 = \{(s_2, \{f\}), (s_3, \{h\})\}$;
$\widehat{C}^{2,1}_1 = \{(s_1, \{b\}), (s_2, \{f\}), (s_3, \{g, h\})\}$;
$\widehat{C}^{2,1}_2 = \{(s_1, \{a, b\}), (s_2, \{c, f\}), (s_3, \{g, h\})\}$;

For the initial state s_1 and initial internal variables $(0, 0, 0, 0)$, only one trajectory available under control of $\widehat{C}$ is $s_1, s_2, s_3, s_2, s_1, ...$, where the sequence of actions is $b, f, g, c, ...$ The trajectory doesn't visit s_4, for the system isn't able to jump to A from s_4 after action e is removed.

4.2 Case study: 1D Hopping Robot

Consider a hopping robot moving in a straight line. The robot model is approximated by Linear Inverted Pendulum Model (LIPM) as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ g/h_0 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ -g/h_0 \end{bmatrix} u \tag{26}$$

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

Table 1: Comparison of re-synthesis time and patching time 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. The last two rows are the time used for re-synthesizing and patching.

U_d	[1]	[1 : 5]	[1 : 10]	[1 : 15]	[1 : 20]	[1 : 25]
∃trans	100%	96.57%	81.22%	60.55%	39.67%	19.00%
$t_{syn}(s)$	244.2	581.2	756.1	532.7	315.1	129.4
$t_{pat}(s)$	3.4	12.6	13.4	13.1	11.4	9.9

Table 2: Comparison of re-synthesis time and patching time 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. The last two rows are the average time used for re-synthesizing and patching.

$\sharp U_d$	1	5	10	15	20	25
$t_{syn}(s)$	289.7	281.1	292.8	358.6	367.1	618.1
$t_{pat}(s)$	3.2	4.1	5.1	7.2	8.3	17.4

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 [5, 12]. Finally we get a AFTS with states indexed $[1 : 4000]$, actions $[1 : 35]$ and 187594 valid transitions, which is the abstraction of the hopping robot.

The specification for the control synthesis is

$$\Diamond\Box B \tag{27}$$

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 (3) (taking $A = Q, B = B, C^1 = Q$).

Now imagine that some holes on the ground are detected, as shown in Figure 4, 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 $U_d = [1], [1 : 5], \ldots, [1 : 25]$ for the hopping robot abstraction and compute the controllers via fixed-point operator (3) and the patching operater (25) respectively. The experiment results in Table 1 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 5a 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.

Furthermore, to show the time efficiency, we randomly choose U_d with size $n = 1, 5, \ldots, 25$, and synthesize the controllers for each U_d via (3) and (25), and then compare the average time used by the two algorithms for each n, shown in Table 2. The time for our patching algorithm is less than 3% of the time for re-synthesizing on average.

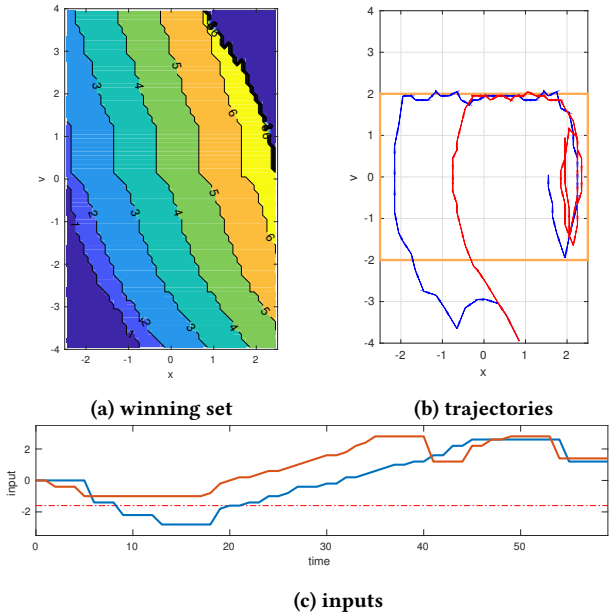

Figure 5: (a) Winning sets under action profiles $U_d = [1], [1 : 5], [1 : 10], \ldots, [1 : 25]$, corresponding to region greater or equal to $1, 2, 3, \ldots, 6$ respectively. (b) Trajectories with initial state $(0.85, -3.95)$ under $U_d = \emptyset$ (blue) and $[1 : 10]$ (red). The region inside 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 input $u = 10$.

(a) is a bit hard to understand: we see only disjoint regions on the image but the winning sets are decreasing in size. Maybe say that the winning set i is the union of the i first regions on the graph. (and redefine them accordingly)

Finally, to show that formal guarantees are satisfied after patching, simulation are run for controllers before and after patching for $U_d = [1 : 10]$. The initial state is $s_0 = 34$. Figure 5b and 5c show the trajectories and the inputs used within 60 time steps. Both trajectories goes into our target region B, indicated by the orange box. The outputs from patched controller is 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 constraints 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. This is actually the motivation of this work.

5 CONCLUSION

In this paper, we proposed a method to do fixed-point based control synthesis by patching the existing controller with a larger action space. The correctness of this method relied on the fact that "nested" structure between fixed-point based controllers with "nested" action

profiles, so we could do the control synthesis by simply finding out the redundant part of the existing controller for the new profile and then removing them. It gives us a way to avoid the expensive computation cost of synthesis from scratch.

We illustrated the efficiency of our method on the example of hopping robot. Given the same specification, the time for patching was only $1.4\% - 7\%$ of the time for synthesis from scratch.

Maybe mention that since in some cases a small change of the action set results in a completely different strategy, we expect that in the worst case, our warm-start method does not bring any speed-up. This is similar to classical warm-up techniques in control: there is no guarantee on the speed-up. And then mention that in future work, although we know that getting formal guarantees on a speed-up in average would be hard (since it highly depends on the type of problem), we wish to conduct a finer study of how much speed-up we can expect to achieve with the proposed warm-up, for instance by studying its effect on "random systems and random specifications

The current work is to apply this method on the control synthesis of hopping robot on a 2D space. If we force the robot to move in 1D and find a finite set of profiles, this method is able to speed up our synthesis process. Besides, it would be interesting to extend this method for the cases that the specification in (1) changes or a set of transitions instead of actions in an AFTS is removed. There is a possibility to do these extension as long as Theorem 1 holds.

REFERENCES

[1] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. 2008. *Principles of model checking*. MIT press.

[2] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv SaÊijar. 2012. Synthesis of reactive (1) designs. *J. Comput. System Sci.* 78, 3 (2012), 911–938.

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

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

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

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

[7] Scott C Livingston, Pavithra Prabhakar, Alex B Jose, and 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, 4588–4595.

[8] Petter Nilsson, Necmiye Ozay, and Jun Liu. 2017. Augmented finite transition systems as abstractions for control synthesis. *Discrete Event Dynamic Systems: Theory and Applications* 27, 2 (jun 2017), 301–340. https://doi.org/10.1007/s10626-017-0243-z

[9] Amir Pnueli and 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, 179–190.

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

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

[12] Fei Sun, Necmiye Ozay, Eric M Wolff, Jun Liu, and 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*. 3273–3280. https://doi.org/10.1109/ACC.2014.6859428

[13] Eric M Wolff, Ufuk Topcu, and 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, 5033–5040.

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