Counter-Strategy Guided Refinement of GR(1) Temporal Logic Specifications

Rajeev Alur, Salar Moarref, and Ufuk Topcu
University of Pennsylvania, Philadelphia, USA. {alur,moarref,utopcu}@seas.upenn.edu

Abstract—**The reactive synthesis problem is to find a finite-state controller that satisfies a given temporal-logic specification regardless of how its environment behaves. Developing a formal specification is a challenging and tedious task and initial specifications are often unrealizable. In many cases, the source of unrealizability is the lack of adequate assumptions on the environment of the system. In this paper, we consider the problem of automatically correcting an unrealizable specification given in the generalized reactivity (1) fragment of linear temporal logic by adding assumptions on the environment. When a temporal-logic specification is unrealizable, the synthesis algorithm computes a counter-strategy as a witness. Our algorithm then analyzes this counter-strategy and synthesizes a set of candidate environment assumptions that can be used to remove the counter-strategy from the environment's possible behaviors. We demonstrate the applicability of our approach with several case studies.**

I. Introduction

Automatically synthesizing a system from a high-level specification is an ambitious goal in the design of reactive systems. The synthesis problem is to find a system that satisfies the specification regardless of how its environment behaves. Therefore, it can be seen as a two-player game between the environment and the system. The environment attempts to violate the specification while the system tries to satisfy it. A specification is *unsatisfiable* if there is no input and output trace that satisfies the specification. A specification is *unrealizable* if there is no system that can implement the specification. That is, the environment can behave in such a way that no matter how the system reacts, the specification would be violated. In this paper we consider specifications which are satisfiable but unrealizable. We address the problem of strengthening the constraints over the environment by adding assumptions in order to achieve realizability.

Writing a correct and complete formal specification which conforms to the (informal) design intent is a hard and tedious task [4], [5]. Initial specifications are often incomplete and unrealizable. Unrealizability of the specification is often due to inadequate environment assumptions. In other words, assumptions about the environment are too weak, leading to an environment with too many behaviors that makes it impossible for the system to satisfy the specification. Usually there is only a rough and incomplete model of the environment in the design phase; thus it is easy to miss assumptions on the environment side. We would like to automatically find such *missing* assumptions that can be added to the specification and make it realizable. Computed assumptions can be used to give the user insight into the specification. They also provide ways to correct the specification. In the context of compositional synthesis [6], [9], derived assumptions based on the components specifications can be used to construct interface rules between the components.

An unrealizable specification cannot be executed or simulated which makes its debugging a challenging task. Counter-strategies are used to explain the reason for unrealizabilty of linear temporal logic (LTL) specifications [5]. Intuitively, a counter-strategy defines how the environment can react to the outputs of the system in order to enforce the system to violate the specification. Konighofer et al. in [5] show how such a counter-strategy can be computed for an unrealizable LTL specification. The requirement analysis tool RATSY [2] implements their method for a fragment of LTL known as generalized reactivity (1) (GR(1)). We also consider GR(1) specifications in this paper because the realizability and synthesis problems for GR(1) specifications can be solved efficiently in polynomial time and GR(1) is expressive enough to be used for interesting real-world problems [3], [12].

Counter-strategies can still be difficult to understand by the user especially for larger systems. We propose a debugging approach which uses the counter-strategies to strengthen the assumptions on the environment in order to make the specification realizable. For a given unrealizable specification, our algorithm analyzes the counter-strategy and synthesizes a set of *candidate* assumptions in the GR(1) form (see section II). Any of the computed candidate assumptions, if added to the specification, restricts the environment in such a way that it cannot behave according to the counter-strategy—without violating its assumptions—anymore. Thus we say the counter-strategy is ruled out from the environment's possible behaviors by adding the candidate assumption to the specification.

The main flow for finding the missing environment assumptions is as follows. If the specification is unrealizable, a counter-strategy is computed for it. A set of *patterns* are then synthesized by processing an abstraction of the counter-strategy. Patterns are LTL formulas of special form that define the structure for the candidate assumptions. We ask the user to specify a set of variables to be used for generating candidates for each pattern. The user can specify the set of variables which she thinks contribute to unrealizability or are underspecified. The variables are used along with patterns to generate the candidate assumptions. Any of the synthesized assumptions can be added to the specification to rule out the counter-strategy. The user can choose an assumption from the candidates in an interactive way or our algorithm can automatically search for it. The chosen assumption is then added to the specification

This research was partially supported by NSF Expedition in Computing project ExCAPE (grant CCF 1138996), and AFOSR (grant number FA9550-12-1-0302).

and the process is repeated with the new specification.

The contributions of this paper are as follows: We propose algorithms to synthesize environment assumptions by directly processing the counter-strategies. We give a counter-strategy guided synthesis approach that finds the missing environment assumptions. The suggested refinement can be validated by the user to ensure compatibility with her design intent and can be added to the specification to make it realizable. We demonstrate our approach with examples and case studies.

The problem of correcting an unrealizable LTL specification by constructing an additional environment assumption is studied by Chatterjee et al. in [4]. They give an algorithm for computing the assumption which only constrains the environment and is as weak as possible. Their approach is more general than ours as they consider general LTL specifications. However, the synthesized assumption is a Büchi automaton which might not translate to an LTL formula and can be difficult for the user to understand (for an example, see Fig. 3 in [4]). Moreover, the resulting specification is not necessarily compatible with the design intent [7]. Our approach generates a set of assumptions in GR(1) form that can easily be validated by the user and be used to make the specification realizable.

The closest work to ours is the work by Li et al. [7] where they propose a template-based specification mining approach to find additional assumptions on the environment that can be used to rule out the counter-strategy. A template is an LTL formula with at least one placeholder, $?_b$, that can be instantiated by the Boolean variable b or its negation. Templates are used to impose a particular structure on the form of generated candidates and are engineered by the user based on her knowledge of the environment. A set of candidate assumptions is generated by enumerating all possible instantiations of the defined templates. For a given counter-strategy, their method finds an assumption from the set of candidate assumptions which is satisfied by the counter-strategy. By adding the negation of such an assumption to the specification, they remove the behavior described by the counter-strategy from the environment. Similar to their work, we consider unrealizable GR(1) specifications and achieve realizability by adding environment assumptions to the specification. But, unlike them, we directly work on the counter-strategies to synthesize a set of candidate assumptions that can be used to rule out the counter-strategy. Similar to templates, patterns impose structure on the assumptions. However, our method synthesizes the patterns based on the counter-strategy and the user does not need to manipulate them. We only require the user to specify a subset of variables to be used in the search for the missing assumptions. The user can specify a subset that she thinks leads to the unrealizability. In our method, the maximum number of generated assumptions for a given counter-strategy is independent from what subset of variables is considered, whereas increasing the size of the chosen subset of variables in [7] will result in exponential growth in the number of candidates, while only a small number of them might hold over all runs of the counter-strategy (unlike our method). Moreover, we compute the weakest environment assumptions for the considered structure and given subset of variables. Our work takes an initial step toward bridging the gap between [4] and [7]. Our method synthesizes environment assumptions that are simple formulas, making them easy to understand and practical, and they also constrain the environment as weakly as possible within their structure. We refer the reader to [7] for a survey of related work.

II. Preliminaries

Linear temporal logic (LTL) is a formal specification language with two kinds of operators: logical connectives (negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$) and implication ($\rightarrow$)) and temporal modal operators (next ($\bigcirc$), always ($\Box$), eventually ($\Diamond$) and until ($\mathcal{U}$)). Given a set P of atomic propositions, an LTL formula is defined inductively as follows: 1) any atomic proposition $p \in P$ is an LTL formula. 2) if ϕ and ψ are LTL formulas, then $\neg\phi$, $\phi \vee \psi$, $\bigcirc\phi$ and $\phi\,\mathcal{U}\,\psi$ are also LTL formulas. Other operators can be defined using the following rules: $\phi \wedge \psi = \neg(\neg\phi \vee \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\Diamond\phi = \mathtt{True}\,\mathcal{U}\,\phi$ and $\Box\phi = \neg\Diamond\neg\phi$. An LTL formula is interpreted over infinite words $\omega \in (2^P)^\omega$. For an LTL formula ϕ, we define its language $\mathcal{L}(\phi)$ to be the set of infinite words that satisfy ϕ, i.e., $\mathcal{L}(\phi) = \{\omega \in (2^P)^\omega \mid \omega \models \phi\}$.

A finite transition system (FTS) is a tuple $\mathcal{T} = \langle Q, Q_0, \delta\rangle$ where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states and $\delta \subseteq Q \times Q$ is the transition relation. An *execution* or *run* of a FTS is an infinite sequence of states $\sigma = q_0q_1q_2...$ where $q_0 \in Q_0$ and for any $i \geq 0$, $q_i \in Q$ and $(q_i, q_{i+1}) \in \delta$. The language of a FTS $\mathcal{T}$ is defined as the set $\mathcal{L}(\mathcal{T}) = \{\omega \in Q^\omega \mid \omega \text{ is a run of } \mathcal{T}\}$, i.e., the set of (infinite) words generated by the runs of $\mathcal{T}$. We often consider a FTS as a directed graph with a natural bijection between the states and transitions of the FTS and vertices and edges of the graph, respectively. Formally for a FTS $\mathcal{T} = \langle Q, Q_0, \delta\rangle$, we define the graph $\mathcal{G}_\mathcal{T} = \langle V, E\rangle$ where each $v_i \in V$ corresponds to a unique state $q_i \in Q$, and $(v_i, v_j) \in E$ if and only if $(q_i, q_j) \in \delta$.

Let P be a set of atomic propositions, partitioned into input, I, and output, O, propositions. A *Moore transducer* is a tuple $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$, where S is the set of states, $s_0 \in S$ is the initial state, $\mathcal{I} = 2^I$ is the input alphabet, $\mathcal{O} = 2^O$ is the output alphabet, $\delta : S \times \mathcal{I} \rightarrow S$ is the transition function and $\gamma : S \rightarrow \mathcal{O}$ is the state output function. A *Mealy* transducer is similar, except that the state output function is $\gamma : S \times \mathcal{I} \rightarrow \mathcal{O}$. For an infinite word $\omega \in \mathcal{I}^\omega$, a run of M is the infinite sequence $\sigma \in S^\omega$ such that $\sigma_0 = s_0$ and for all $i \geq 0$ we have $\sigma_{i+1} = \delta(\sigma_i, \omega_i)$. The run σ on input word ω produces an infinite word $M(\omega) \in (2^P)^\omega$ such that $M(\omega)_i = \gamma(\sigma_i) \cup \omega_i$ for all $i \geq 0$. The language of M is the set $\mathcal{L}(M) = \{M(\omega) \mid \omega \in \mathcal{I}^\omega\}$ of infinite words generated by runs of M.

An LTL formula ϕ is *satisfiable* if there exists an infinite word $\omega \in (2^P)^\omega$ such that $\omega \models \phi$. A Moore (Mealy) transducer M satisfies an LTL formula ϕ, written as $M \models \phi$, if $\mathcal{L}(M) \subseteq \mathcal{L}(\phi)$. An LTL formula ϕ is *Moore (Mealy) realizable* if there exists a Moore (Mealy, respectively) transducer M such that $M \models \phi$. The *realizability problem* asks whether there exists such a transducer for a given LTL specification ϕ.

A *two-player deterministic game graph* is a tuple $\mathcal{G} = (Q, Q_0, E)$ where Q can be partitioned into two disjoint sets Q_1 and Q_2. Q_1 and Q_2 are the sets of states of player 1 and 2, respectively. Q_0 is the set of initial states. $E = Q \times Q$ is the set of directed edges. Players take turns to play the game. At each step, if the current state belongs to Q_1, player 1 chooses the next state. Otherwise player 2 makes a move. A *play* of the game graph $\mathcal{G}$ is an infinite sequence $\sigma = q_0q_1q_2...$ of states such that $q_0 \in Q_0$, and $(q_i, q_{i+1}) \in E$ for all $i \geq 0$. We denote the set of all plays by Π. A *strategy* for player $i \in \{1, 2\}$ is a function $\alpha_i : Q^*.Q_i \to Q$ that chooses the next state given a finite sequence of states which ends at a player i state. A strategy is *memoryless* if it is a function of current state of the play, i.e., $\alpha_i : Q_i \to Q$. Given strategies α_1 and α_2 for players and a state $q \in Q$, the *outcome* is the play starting at q, and evolved according to α_1 and α_2. Formally, $outcome(q, \alpha_1, \alpha_2) = q_0q_1q_2...$ where $q_0 = q$, and for all $i \geq 0$ we have $q_{i+1} = \alpha_1(q_0q_1...q_i)$ if $q_i \in Q_1$ and $q_{i+1} = \alpha_2(q_0q_1...q_i)$ if $q_i \in Q_2$. An *objective* for a player is a set $\Phi \subseteq \Pi$ of plays. A strategy α_1 for player 1 is winning for some state q if for every strategy α_2 of player 2, we have $outcome(q, \alpha_1, \alpha_2) \in \Phi$.

Given an LTL formula ϕ over P and a partitioning of P into I and O, the *synthesis problem* is to find a Mealy transducer M with input alphabet $\mathcal{I} = 2^I$ and output alphabet $\mathcal{O} = 2^O$ that satisfies ϕ. This problem can be reduced to computing winning strategies in game graphs. A deterministic game graph G, and an objective Φ can be constructed such that ϕ is realizable if and only if the system (player 1) has a memoryless winning strategy from the initial state in G [11]. Every memoryless winning strategy of the system can be represented by a Mealy transducer that satisfies ϕ. If the specification ϕ is unrealizable, then the environment (player 2) has a winning strategy. A *counter-strategy* for the synthesis problem is a strategy for the environment that can falsify the specification, no matter how the system plays. Formally, a counter-strategy can be represented by a Moore transducer $M_c = (S', s'_0, \mathcal{I}', \mathcal{O}', \delta', \gamma')$ that satisfies $\neg\phi$, where $\mathcal{I}' = \mathcal{O}$ and $\mathcal{O}' = \mathcal{I}$ are the input and output alphabet for M_c which are generated by the system and the environment, respectively.

In this paper, we consider specifications of the form

$$\phi = \phi_e \to \phi_s, \tag{1}$$

where ϕ_α for $\alpha \in \{e, s\}$ can be written as a conjunction of the following parts:

- ϕ_i^α: A Boolean formula over I if $\alpha = e$ and over $I \cup O$ otherwise, characterizing the initial state.
- ϕ_t^α: An LTL formula of the form $\bigwedge_i \Box\psi_i$. Each subformula $\Box\psi_i$ is either characterizing an invariant, in which case ψ_i is a Boolean formula over $I \cup O$, or it is characterizing a transition relation, in which case ψ_i is a Boolean formula over expressions v and $\bigcirc v'$ where $v \in I \cup O$ and, $v' \in I$ if $\alpha = e$ and $v' \in I \cup O$ if $\alpha = s$.
- ϕ_g^α: A formula of the form $\bigwedge_i \Box\Diamond B_i$ characterizing fairness/liveness, where each B_i is a Boolean formula over $I \cup O$.

For the specifications of the form in (1), known as GR(1) formulas, Piterman et al. [10] show that the synthesis problem can be solved in polynomial time. Intuitively, in (1), ϕ_e characterizes the assumptions on the environment and ϕ_s characterizes the correct behavior (guarantees) of the system. Any correct implementation of the specification guarantees to satisfy ϕ_s, provided that the environment satisfies ϕ_e.

For a given unrealizable specification $\phi_e \to \phi_s$, we define a *refinement* $\psi = \bigwedge_i \psi_i$ as a conjunction of a collection of environment assumptions ψ_i in the GR(1) form such that $\phi_e \wedge \psi \to \phi_s$ is realizable. Intuitively it means that adding the assumptions ψ_i to the specification results in a new specification which is realizable. We say a refinement ψ is consistent with the specification $\phi_e \to \phi_s$ if $\phi_e \wedge \psi$ is satisfiable. Note that if $\phi_e \wedge \psi$ is not satisfiable, i.e., $\phi_e \wedge \psi = \texttt{False}$, the specification $\phi_e \wedge \psi \to \phi_s$ is trivially realizable [7], but obviously ψ is not an interesting refinement.

III. Problem Statement and Overview

A. Problem Statement

Given a specification $\phi = \phi_e \to \phi_s$ in the GR(1) form which is satisfiable but unrealizable, find a refinement $\psi = \bigwedge_i \psi_i$ as a conjunction of environment assumptions ψ_i such that $\phi_e \wedge \psi$ is satisfiable and $\phi_e \wedge \psi \to \phi_s$ is realizable.

B. Overview of the Method

We now give a high-level view of our method. Specification refinements are constructed in two phases. First, given a counter-strategy's Moore machine M_c, we build an abstraction which is a FTS $\mathcal{T}_c$. The abstraction preserves the structure of the counter-strategy (its states and transitions) while removing the input and output details. The algorithm processes $\mathcal{T}_c$ and synthesizes a set of LTL formulas in special forms, called *patterns*, which hold over *all* runs of $\mathcal{T}_c$. Our algorithm then uses these patterns along with a subset of variables specified by the user to generate a set of LTL formulas which hold over *all* runs of M_c. We ask the user to specify a subset of variables which she thinks contribute to the unrealizability of the specification. This set can also be used to guide the algorithm to generate formulas over the set of variables which are underspecified. Using a smaller subset of variables leads to simpler formulas that are easier for the user to understand.

The complement of the generated formulas form the set of candidate assumptions that can be used to rule out the counter-strategy from the environment's possible behaviors. We remove the candidates which are not consistent with the specification in order to avoid a trivial solution `False`.

Any assumption from the set of generated candidates can be used to rule out the counter-strategy. Our approach does a breadth-first search over the candidates. If adding any of the candidates makes the specification realizable, the algorithm returns that candidate as a solution. Otherwise at each iteration, the process is repeated for any of the new specifications resulting from adding a candidate. The depth of the search is controlled by the user. The search continues until either a consistent refinement is found or the algorithm cannot find one within the specified depth (hence the search algorithm is sound, but not complete).

Example 1. *Consider the following example borrowed from [7] with the environment variables $I = \{r, c\}$ and system variables $O = \{g, v\}$. Here r, c, g and v stand for* request, clear, grant *and* valid *signals respectively. We start with no assumption, that is we only assume $\phi_e = \texttt{True}$. Consider the following system guarantees: $\phi_1 = \Box(r \to \bigcirc\Diamond g)$, $\phi_2 = \Box((c \vee g) \to \bigcirc\neg g)$, $\phi_3 = \Box(c \to \neg v)$ and $\phi_4 = \Box\Diamond(g \wedge v)$.*

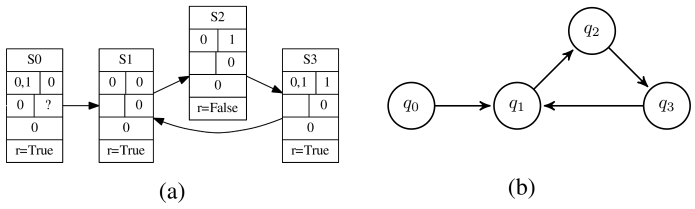

Fig. 1: (a) A counter-strategy produced by RATSY for the specification of Example 1 with the additional assumption $\Box\Diamond(\neg r)$. $c = \texttt{True}$ is constant in all states. (b) The abstract finite transition system for the counter-strategy of part (a).

Let ϕ_s be the conjunction of these formulas. ϕ_1 requires that every request must be granted eventually starting from the next step by setting signal g to high. ϕ_2 says that if clear or grant signal is high, then grant must be low at the next step. ϕ_3 says if clear is high, then the valid signal must be low. Finally, ϕ_4 says that system must issue a valid grant infinitely often.

The specification $\phi_e \rightarrow \phi_s$ is unrealizable. A simple counter-strategy is for the environment to keep r and c high at all times. Then, by ϕ_3, v needs to be always low and thus ϕ_4 cannot be satisfied by any system. RATSY produces this counter-strategy which is then fed to our algorithm. An example candidate found by our algorithm to rule out this counter-strategy is the assumption $\psi = \Box\Diamond(\neg r)$. Adding ψ to the specification prevents the environment from always keeping r high, thus the environment cannot use the counter-strategy anymore. However, the specification $\phi_e \wedge \psi \rightarrow \phi_s$ is still unrealizable. RATSY produces the counter-strategy shown in Figure 1(a) for the new specification. The new counter-strategy keeps the c high all the times. The value of r is changed depending on the state of the counter-strategy as shown in Figure 1(a). The top block in each state of Figure 1(a) is the name of the state. RATSY produces additional information, shown in middle blocks, on how the counter-strategy enforces the system to violate the specification. We do not use this information in the current version of the algorithm.

The following formulas are examples of consistent refinements produced by our algorithm for the specification $\phi_e \rightarrow \phi_s$:

- $\psi_1 = \Box(\neg r \vee \neg c) \wedge \Box(r \vee \neg c)$
- $\psi_2 = \Box(r \rightarrow \bigcirc\neg c) \wedge \Box(\neg r \rightarrow \bigcirc\neg c)$
- $\psi_3 = \Box\Diamond(\neg r) \wedge \Box(\neg c \vee r) \wedge \Box(\neg r \rightarrow \bigcirc\neg c))$

Assumptions in both of the refinements ψ_1 and ψ_2 imply $\Box(\neg c)$, that is, adding them requires the environment to keep the signal c always low. Although adding these assumptions make the specification realizable, it may not conform to the design intent. Refinement ψ_3 does not restrict c like ψ_1 and ψ_2, and only assumes that the environment sets the signal r to low infinitely often and that, when the request signal is low, the clear signal should be low at the same and the next step.

IV. Specification Refinement

Algorithm 1 finds environment assumptions that can be added to the specification to make it realizable. It gets as input the initial unrealizable specification $\phi = \phi_e \rightarrow \phi_s$, the set P of subsets of variables to be used in generated assumptions and the maximum depth α of the search. It outputs a consistent refinement ψ, if it can find one within the specified depth.

For an unrealizable specification, a counter-strategy is computed as a Moore transducer using the techniques in [5], [2]. The counter-strategy is then fed to the **GeneratePatterns** procedure which constructs a set of patterns and is detailed in Section IV-C. Procedure **GenerateCandidates**, described in Section IV-A, produces a set of candidate assumptions in the form of GR(1) formulas using patterns and the set P of variables. Algorithm 1 runs a breadth-first search to find a consistent refinement. Each node of the search tree is a generated candidate assumption, while the root of the tree corresponds to the assumption `True` (i.e., no assumption). Each path of the search tree starting from the root corresponds to a candidate refinement as conjunction of candidate assumptions of the nodes visited along the path. When a node is visited during the search, its corresponding candidate refinement is added to the specification. If the new specification is consistent and realizable, the refinement is returned by the algorithm. Otherwise, if the depth of the current node is less than the maximum specified, a set of candidate assumptions are generated based on the counter-strategy for the new specification and the search tree expands.

In Algorithm 1, the queue *CandidatesQ* keeps the candidate refinements which are found during the search. At each iteration, a candidate refinement ψ is removed from the head of the queue. The procedure **Consistent** checks if ψ is consistent with the specification ϕ. If it is, the algorithm checks the realizability of the new specification $\phi_{new} = \phi_e \wedge \psi \rightarrow \phi_s$ using the procedure **Realizable** [3], [2]. If ϕ_{new} is realizable, ψ is returned as a suggested refinement. Otherwise, if the depth of the search for reaching the candidate refinement ψ is less than α, a new set of candidate assumptions are generated using the counter-strategy computed for ϕ_{new}. Algorithm 1 keeps track of the number of counter-strategies produced along the path to reach a candidate refinement in order to compute its depth (**Depth**(ψ)). Each new candidate assumption ψ_{new} results in a new candidate refinement $\psi \wedge \psi_{new}$ which is added to the end of the queue for future processing . The algorithm terminates when either a consistent refinement ψ is found, or there is no more candidates in the queue to be processed.

A. Generating Candidates

Consider the Moore transducer $M_c = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \gamma)$ of a counter-strategy, where $\mathcal{I} = 2^O$ and $\mathcal{O} = 2^I$, and O and I are the set of the system and environment variables, respectively. Given M_c, we construct a finite transition system $\mathcal{T}_c = \langle Q, \{q_0\}, \delta \rangle$ which preserves the structure of the M_c while removing all details about its input and output. More formally, for each state $s_i \in S$, $\mathcal{T}_c$ has a corresponding state $q_i \in Q$, and $q_0 \in Q$ is the state corresponding to $s_0 \in S$. There exists a transition $(q_i, q_j) \in \delta$ if and only if there exists $y \in \mathcal{I}$ such that $\delta(s_i, y) = s_j$. It is easy to see that any run of $\mathcal{T}_c$ corresponds to a run of M_c and vice versa.

By processing the abstract FTS $\mathcal{T}_c$ of the counter-strategy, we synthesize a set of patterns which are LTL formulas of the form $\Diamond\Box\psi_1$, $\Diamond\psi_2$ and $\Diamond(\psi_3 \wedge \bigcirc\psi_4)$ that hold over all runs of $\mathcal{T}_c$. Each ψ_i for $i \in \{1, 2, 3, 4\}$ is a disjunction of a subset of states of $\mathcal{T}_c$, i.e., $\psi_i = \bigvee_{q \in Q_i} q$ where $Q_i \subseteq Q$.

Algorithm 1: Specification Refinement

```
Input: $\phi = \phi_e \rightarrow \phi_s$, initial specification
Input: $P$, set of subsets of variables to be used in
       patterns
Input: $\alpha$, maximum depth of the search
Output: $\psi$, additional assumptions such that
        $\phi_e \wedge \psi \rightarrow \phi_s$ is realizable
1  $M_c$ := CounterStrategy($\phi$);
2  Patterns := GeneratePatterns($M_c$);
3  CandidatesQ := GenerateCandidates(Patterns,$P$);
4  while CandidatesQ is not Empty do
5      $\psi$ := CandidatesQ.DeQueue;
6      if Consistent($\phi$,$\psi$) then
7          $\phi_{new} = \phi_e \wedge \psi \rightarrow \phi_s$;
8          if Realizable($\phi_{new}$) then
9              return $\psi$;
10         else
11             if Depth($\psi$) $< \alpha$ then
12                 $M_c$ := CounterStrategy($\phi_{new}$);
13                 Patterns := GeneratePatterns($M_c$);
14                 newCandidates :=
                   GenerateCandidates(Patterns,$P$) ;
15                 foreach $\psi_{new} \in$ newCandidates do
16                     CandidatesQ.EnQueue($\psi \wedge \psi_{new}$);
17 return No refinement was found;
```

The complements of these formulas, $\Box\Diamond\neg\psi_1$ (liveness), $\Box\neg\psi_2$ (safety), and $\Box(\psi_3 \rightarrow \bigcirc\neg\psi_4)$ (transition), respectively, are of the desired GR(1) form and provide the structure for the candidate assumptions that can be used to rule out the counter-strategy. Note that similar to [7], we do not synthesize assumptions characterizing the initial state because they are easy to specify in practice. Besides, it is simple to discover them from the counter-strategy. Patterns are generated using simple graph search algorithms explained in Section IV-C.

Example 2. *Figure 1(b) shows the abstract FTS for the counter-strategy of Figure 1(a). For this FTS our algorithm produces the set of patterns* $\Diamond\Box(q_1 \vee q_2 \vee q_3)$, $\Diamond q_0, \Diamond q_1, \Diamond q_2, \Diamond q_3$, *and* $\Diamond(q_0 \wedge \bigcirc q_1)$, $\Diamond(q_1 \wedge \bigcirc q_2)$, $\Diamond(q_2 \wedge \bigcirc q_3)$, $\Diamond(q_3 \wedge \bigcirc q_1)$. *Any run of* $\mathcal{T}_c$ *satisfies all of the above formulas. For example* $\mathcal{T}_c \models \Diamond q_i$ *for* $i \in \{0,1,2,3\}$, *meaning that any run of the* $\mathcal{T}_c$ *will eventually visit state* q_i. *The formula* $\Diamond(q_1 \wedge \bigcirc q_2)$ *means that any run of* $\mathcal{T}_c$ *will eventually visit state* q_1 *and then state* q_2 *at the next step. Also any run of* $\mathcal{T}_c$ *satisfies* $\Diamond\Box(q_1 \vee q_2 \vee q_3)$, *meaning that any run of* $\mathcal{T}_c$ *will eventually reach and stay in the set of states* $\{q_1, q_2, q_3\}$.

As we mentioned previously, each state $q_i \in Q$ of the FTS $\mathcal{T}_c$ corresponds to a state $s_i \in S$ of the Moore transducer M_c of the counter-strategy. Also recall that each run of $\mathcal{T}_c$ corresponds to a run of M_c. M_c, at any state $s_i \in S$, outputs the propositional formula $\mathcal{V}_{s_i} = \gamma(s_i)$ which is a valuation over all environment variables. Formally, for any state $s_i \in S$ of M_c, we have $\mathcal{V}_{s_i} = \ell_1^i \wedge \ell_2^i \wedge ... \wedge \ell_n^i$ where each ℓ_j^i is a literal over the environment variable $x_j \in I$. We call $\mathcal{V}_{s_i}$ the state predicate of s_i and also q_i. We replace the states in the patterns with their corresponding state predicates to get a set of formulas which hold over all runs of the counter-strategy.

Example 3. *Consider the counter-strategy shown in Figure 1(a). The state predicates are* $\mathcal{V}_{S0} = \mathcal{V}_{S1} = \mathcal{V}_{S3} = c \wedge r$ *and* $\mathcal{V}_{S2} = c \wedge \neg r$, *where* $S0, S1, S2$ *and* $S3$ *are the states of* M_c. *Using the patterns obtained in Example 2 and replacing the states with their corresponding state predicates, we obtain LTL formulas which hold over all runs of* M_c. *For example, the pattern* $\Diamond\Box(q_1 \vee q_2 \vee q_3)$ *gives us the formula* $\Diamond\Box((c \wedge r) \vee (c \wedge \neg r)) = \Diamond\Box c$. *Replacing* q_2 *with* $\mathcal{V}_{S2}$ *in the pattern* $\Diamond q_2$ *leads to* $\Diamond(c \wedge \neg r)$. *Similarly, the pattern* $\Diamond(q_1 \wedge \bigcirc q_2)$ *gives* $\Diamond((c \wedge r) \wedge \bigcirc(c \wedge \neg r))$.

The structure of the state predicates and patterns is such that any subset of the environment variables can be used along with the patterns to generate candidates and the resulting formulas still hold over all runs of the counter-strategy. Algorithm 1 gets the set $P = \{P_1, P_2, P_3, P_4\}$ as input, where each P_i is a subset of environment variables that should be used in the corresponding ψ_i for generating the candidate assumptions from the patterns of the form $\Diamond\Box\psi_1$, $\Diamond\psi_2$ and $\Diamond(\psi_3 \wedge \bigcirc\psi_4)$.

Example 4. *Assume that the designer specifies* $P_1 = \{r\}$, $P_2 = \{c\}$, $P_3 = \{r, c\}$ *and* $P_4 = \{c\}$. *Then the pattern* $\Diamond\Box(q_1 \vee q_2 \vee q_3)$ *results in* $\Diamond\Box(r \vee \neg r \vee r) = \Diamond\Box\mathtt{True}$. *From* $\Diamond q_2$ *we obtain* $\Diamond c$, *and* $\Diamond(q_1 \wedge \bigcirc q_2)$ *leads to* $\Diamond((c \wedge r) \wedge \bigcirc c)$. *Note that using a smaller subset of variables leads to simpler formulas (and sometimes trivial as in* $\Diamond\Box(\mathtt{True})$*). However, this simplicity may result in assumptions which put more constraints on the environment as we will show later.*

The complement of the generated formulas form the set of candidate assumptions that can be used to rule out the counter-strategy. For instance, formulas $\Box\Diamond(\neg r \wedge r) = \Box\Diamond(\mathtt{False})$, $\Box(\neg c)$, $\Box((c \wedge r) \rightarrow \bigcirc(\neg c))$ and $\Box((c \wedge \neg r) \rightarrow \bigcirc(\neg c))$ are the candidate assumptions computed based on the user input in Example 4. Note that there might be repetitive formulas among the generated candidates. We remove the repeated formulas in order to prevent the process from checking the same assumption repeatedly. We also use some techniques to simplify the synthesized assumptions (see [1]).

B. Removing the Restrictive Formulas

Given two non-equivalent formulas ϕ_1 and ϕ_2 we say ϕ_1 is *stronger* than ϕ_2 if $\phi_1 \rightarrow \phi_2$ holds. Assume ψ_1 and ψ_2 are two formulas that hold over all runs of the counter-strategy computed for the specification $\phi_e \rightarrow \phi_s$, and that $\psi_1 \rightarrow \psi_2$. Note that $\neg\psi_2 \rightarrow \neg\psi_1$ also holds, that is $\neg\psi_1$ is a *weaker* assumption compared to $\neg\psi_2$. Adding either $\neg\psi_1$ or $\neg\psi_2$ to the environment assumptions ϕ_e rules out the counter-strategy. However, adding the stronger assumption $\neg\psi_2$ restricts the environment more than adding $\neg\psi_1$. That is, $\phi_e \wedge \neg\psi_2$ puts more constraints on the environment compared to $\phi_e \wedge \neg\psi_1$.

As an example, consider the counter-strategy M_c shown in Figure 1(a). Both $\psi_1 = \Diamond(c \wedge \neg r)$ and $\psi_2 = \Diamond(c)$ hold over all runs of M_c. Moreover, $\psi_1 \rightarrow \psi_2$. Consider the corresponding assumptions $\neg\psi_1 = \Box(\neg c \vee r)$ and $\neg\psi_2 = \Box(\neg c)$. Adding $\neg\psi_2$ restricts the environment more than adding $\neg\psi_1$. $\neg\psi_2$ requires the environment to keep the signal c always low, whereas in case of $\neg\psi_1$, the environment is free to assign additional values to its variables. It only prevents the environment from setting c to high and r to low at the same time.

We construct patterns which are strongest formulas of their specified form that hold over all runs of the counter-strategy. Therefore, the generated candidate assumptions are the weakest formulas that can be constructed for the given structure and the user specified subset of variables.

C. Synthesizing Patterns

In this section we show how certain types of patterns can be synthesized using the abstract FTS $\mathcal{T}_c$ of the counter-strategy. A pattern $\mathcal{P}$, is an LTL formula $\phi_{\mathcal{P}}$ which holds over all runs of the FTS $\mathcal{T}_c$, i.e., $\mathcal{T}_c \models \phi_{\mathcal{P}}$. We are interested in patterns of the form $\Diamond\Box\psi$, $\Diamond\psi$ and $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$. The complements of these patterns are of the GR(1) form and, after replacing states with their corresponding state predicates, will yield to candidate assumptions for removing the counter-strategy.

1) Patterns of the Form $\Diamond\psi$: For a FTS $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$, we define a *configuration* $C \subseteq Q$ as a subset of states of $\mathcal{T}_c$. We say a configuration C is an *eventually configuration* if for any run σ of $\mathcal{T}_c$ there exists a state $q \in C$ and a time step $i \geq 0$ such that $\sigma_i = q$. That is, any run of $\mathcal{T}_c$ eventually visits a state from the configuration C. It follows that if C is an eventually configuration for $\mathcal{T}_c$, then $\mathcal{T}_c \models \Diamond \bigvee_{q\in C} q$. We say an eventually configuration C is *minimal* if there exists no $C' \subset C$ such that C' is an eventually configuration. Note that removing any state $q \in C$ from a minimal eventually configuration leads to a configuration which is not an eventually configuration.

Algorithm 2 constructs eventually patterns which correspond to the minimal eventually configurations of $\mathcal{T}_c$ with size less than or equal to β. The larger configurations lead to larger formulas which are hard for the user to parse. The user can specify the value of β. Heuristics can also be used to automatically set β based on the properties of $\mathcal{T}_c$, e.g. the maximum outdegree of the vertices in the corresponding directed graph $\mathcal{G}_{\mathcal{T}_c}$, where the outdegree of a vertex is the number of its outgoing edges. In Algorithm 2, the set $\Diamond$Configurations keeps the minimal eventually configurations discovered so far. Algorithm 2 initializes the sets Patterns and $\Diamond$Configurations to $\{\Diamond q_0\}$ and $\{q_0\}$, respectively. Note that $\Diamond q_0$ holds over all runs of $\mathcal{T}_c$. The algorithm then checks each possible configuration $Q' \subseteq Q - \{q_0\}$ with size less than or equal to β in a non-decreasing order of $|Q'|$ to find minimal eventually configurations. Without loss of generality we assume that all states in $\mathcal{T}_c$ have outgoing edges[1]. At each iteration, a configuration Q' is chosen. Algorithm 2 checks if there is a minimal eventually configuration Q'' which is already discovered and $Q'' \subset Q'$. If such Q'' exists, Q' is not minimal. Otherwise, the algorithm checks if it is an eventually configuration by first removing all the states in Q' and their corresponding incoming and outgoing transitions from $\mathcal{T}_c$ to obtain another FTS $\mathcal{T}_c'$. Now, if there is an infinite run from q_0 in $\mathcal{T}_c'$, then there is a run in $\mathcal{T}_c$ that does not visit any state in Q'. Otherwise, Q' is a minimal eventually configuration and is added to $\Diamond$Configurations. The corresponding formula $\psi = \Diamond \bigvee_{q\in Q'} q$ is also added to the set of eventually patterns. Note that checking if there exists an infinite run in $\mathcal{T}_c'$ can be done by considering $\mathcal{T}_c'$ as a graph and checking if there is a reachable cycle from q_0, which can be done in linear time in number of states and transitions of $\mathcal{T}_c$. Therefore, the algorithm is of complexity $O(|Q|^{\beta}(|Q| + |\delta|))$.

[1]A transition from any state with no outgoing transition can be added to a dummy state with a self loop. Patterns which include the dummy state will be removed.

Algorithm 2: Generating $\Diamond\psi$ patterns

```
   Input: Finite state transition system $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$
   Input: $\beta$, maximum number of states in generated patterns
   Output: a set of patterns of the form $\Diamond\psi$ where $\mathcal{T}_c \models \Diamond\psi$
1  Patterns := $\{\Diamond q_0\}$;
2  $\Diamond$Configurations := $\{q_0\}$;
3  foreach $Q' \subseteq Q - \{q_0\}$ with non-decreasing order of $|Q'|$ where $|Q'| \leq \beta$ do
4      if $\nexists Q'' \in \Diamond$Configurations s.t. $Q'' \subseteq Q'$ then
5          Let $\mathcal{T}_c' = \langle Q - Q', \{q_0\}, \delta'\rangle$ where $\delta' = \{(q, q') \in \delta | q \notin Q' \wedge q' \notin Q'\}$;
6          if there is no infinite run from $q_0$ in $\mathcal{T}_c'$ then
7              Add $Q'$ to $\Diamond$Configurations;
8              Let $\psi = \Diamond \bigvee_{q_i \in Q'} q_i$;
9              Add $\psi$ to Patterns;
10 return Patterns;
```

Example 5. *Consider the FTS shown in Figure 2. Algorithm 2 starts at initial configuration $\{q_0\}$ and generates the formula $\Diamond q_0$. None of $\{q_1\}$, $\{q_2\}$ or $\{q_3\}$ is an eventually configuration. For example for configuration $\{q_1\}$, there exists the run $\sigma = q_0, (q_3)^{\omega}$ which never visits q_1. Configurations $\{q_1, q_3\}$ and $\{q_2, q_3\}$ are minimal eventually configurations. For example removing $\{q_1, q_3\}$ will lead to a FTS with no infinite run (no cycle is reachable from q_0 in the corresponding graph). It is easy to see that configuration $\{q_1, q_2\}$ is not an eventually configuration. Configuration $\{q_1, q_2, q_3\}$ is not minimal, although it is an eventually configuration. Thus Algorithm 2 returns the set of patterns $\{\Diamond q_0, \Diamond(q_1 \vee q_3), \Diamond(q_2 \vee q_3)\}$.*

2) Patterns of the Form $\Diamond\Box\psi$: To compute formulas of the form $\Diamond\Box\psi$ which hold over all runs of the FTS $\mathcal{T}_c = \langle Q, \{q_0\}, \delta\rangle$ of the counter-strategy, we view $\mathcal{T}_c$ as a graph and separate its states into two groups: $Q^{cycle} \subseteq Q$, the set of states that are part of a cycle in $\mathcal{T}_c$ (including the cycle from one node to itself), and $Q' = Q - Q^{cycle}$. Without loss of generality we assume that any state $q \in Q$ is reachable from q_0. Therefore, any state $q \in Q^{cycle}$ belongs to a reachable strongly connected component C of $\mathcal{T}_c$. Also for any strongly connected component C of $\mathcal{T}_c$, there exists a run σ of $\mathcal{T}_c$ which reaches states in C and keeps cycling there forever. Hence, the formula $\psi_1 = \Diamond\Box \bigvee_{q\in C} q$ holds over the run σ. Indeed ψ_1 is the minimal formula of disjunctive form which holds over all runs that can reach the strongly connected component C. That is, by removing any of the states from ψ_1, one can find a run σ' which can reach the strongly connected component C and visit the removed state, falsifying the resulted formula. Therefore, eventually for any execution of $\mathcal{T}_c$, the state of the system will always be in one of the states $q \in Q^{cycle}$. Thus the formula $\psi = \Diamond\Box \bigvee_{q\in Q^{cycle}} q$ is the minimal formula of the form eventually always which holds over all runs of $\mathcal{T}_c$.

To partition the states of the $\mathcal{T}_c$ into Q^{cycle} and Q' we use Tarjan's algorithm for computing strongly connected components of the graph. Thus the algorithm is of linear time

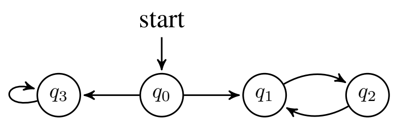

Fig. 2: A non-deterministic finite state transition system $\mathcal{T}_c$

complexity in number of states and transitions of $\mathcal{T}_c$.

Example 6. *Consider the non-deterministic FTS shown in Figure 2. It has three strongly connected components: $\{q_0\}$, $\{q_1, q_2\}$ and $\{q_3\}$. Only the latter two components include a cycle inside them, that is $Q^{cycle} = \{q_1, q_2, q_3\}$. Thus, the pattern $\psi = \Diamond\Box(q_1 \vee q_2 \vee q_3)$ is generated. Note that the possible runs of the system are $\sigma_1 = q_0, (q_1, q_2)^\omega$ and $\sigma_2 = q_0, (q_3)^\omega$. The generated pattern ψ holds over both of these runs. Observe that removing any of the states in ψ will result in a formula which is not satisfied by $\mathcal{T}_c$ any more.*

3) Patterns of the Form $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$: To generate candidates of the form $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$, first note that $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$ holds only if $\Diamond\psi_1$ holds. Therefore, a set of eventually patterns $\Diamond\psi_1$ is first computed using Algorithm 2. Then for each formula $\Diamond\psi_1$, the pattern $\Diamond(\psi_1 \wedge \bigcirc \bigvee_{q \in Next(\psi_1)} q)$ is generated, where $Next(\psi_1)$ is the set of states that can be reached in one step from the configuration specified by ψ_1. Formally, $Next(\psi_1) = \{q_i \in Q \mid \exists q_j \in \mathcal{C} \text{ s.t. } (q_j, q_i) \in \delta\}$ and $\mathcal{C}$ is the configuration represented by $\psi_1 = \bigvee_{q \in \mathcal{C}} q$. The most expensive part of this procedure is computing the eventually patterns, therefore its complexity is the same as Algorithm 2. Algorithms for computing $\Diamond\Box\psi$ and $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$ patterns can be found in the technical report [1].

Example 7. *Consider the FTS shown in Figure 2. Given the set of eventually formulas produced in Example 5, patterns $\Diamond(q_0 \wedge \bigcirc(q_1 \vee q_3))$, $\Diamond((q_1 \vee q_3) \wedge \bigcirc(q_2 \vee q_3))$ and $\Diamond((q_2 \vee q_3) \wedge \bigcirc(q_1 \vee q_3))$ are generated.*

The procedures described for producing patterns, lead to assumptions which only include environment variables, and are enough for resolving unrealizability in our case studies. However, in general, GR(1) assumptions can also include the system variables. The procedures can be easily extended to the general case (see [1]).

The following theorem states that the procedures described in this section, generate the strongest patterns of the specified forms. Its proof can be found in [1]. Removing the weaker patterns leads to shorter formulas which are easier for the user to understand. It also decreases the number of generated candidates at each step. More importantly, it leads to weaker assumptions on the environment that can be used to rule out the counter-strategy. If the restriction imposed by any of these candidates is not enough to make the specification realizable, the method analyzes the counter-strategy computed for the new specification to find assumptions that can restrict the environment more. This way the counter-strategies guide the method to synthesize assumptions that can be used to achieve realizability.

Theorem 1. *For any formula of the form $\Diamond\psi$, $\Diamond\Box\psi$, or $\Diamond(\psi_1 \wedge \bigcirc\psi_2)$ which hold over all runs of a given FTS $\mathcal{T}_c$, there is an equivalent or stronger formula of the same form synthesized by the algorithms described in Section IV-C.*

V. Case studies

We now present two case studies. We use RATSY to generate counter-strategies and Cadence SMV model checker [8] to check the consistency of the generated candidates. In our experiments, we set α in Algorithm 1 to two, and β in Algorithm 2 to the maximum outdegree of the vertices of the counter-strategy's abstract directed graph. We slightly change Algorithm 1 to find all possible refinements within the specified depth.

A. Lift Controller

We borrow the lift controller example from [3]. Consider a lift controller serving three floors. Assume that the lift has three buttons, denoted by the Boolean variables b_1, b_2 and b_3, which are controlled by the environment. The location of the lift is represented using Boolean variables f_1, f_2 and f_3 controlled by the system. The lift may be requested on each floor by pressing the corresponding button. We assume that (1) once a request is made, it cannot be withdrawn, (2) once the request is fulfilled it is removed, and (3) initially there are no requests. Formally, the specification of the environment is $\phi_e = \phi^e_{init} \wedge \phi^e_{1_1} \wedge \phi^e_{1_2} \wedge \phi^e_{1_3} \wedge \phi^e_{2_1} \wedge \phi^e_{2_2} \wedge \phi^e_{2_3}$, where $\phi^e_{init} = (\neg b_1 \wedge \neg b_2 \wedge \neg b_3)$, $\phi^e_{1_i} = \Box(b_i \wedge f_i \rightarrow \bigcirc\neg b_i)$, and $\phi^e_{2_i} = \Box(b_i \wedge \neg f_i \rightarrow \bigcirc b_i)$ for $1 \leq i \leq 3$.

The lift initially starts on the first floor. We expect the lift to be only on one of the floors at each step. It can move at most one floor at each time step. We want the system to eventually fulfill all the requests. Formally the specification of the system is given as $\phi_s = \phi^s_{init} \wedge \phi^s_1 \bigwedge_i \phi^s_{2,i} \wedge \phi^s_3 \bigwedge_j \phi^s_{4,j} \wedge \phi^s_5$, where

- $\phi^s_{init} = f_1 \wedge \neg f_2 \wedge \neg f_3$,
- $\phi^s_1 = \Box(\neg(f_1 \wedge f_2) \wedge \neg(f_2 \wedge f_3) \wedge \neg(f_1 \wedge f_3))$,
- $\phi^s_{2,i} = \Box(f_i \rightarrow \bigcirc(f_{i-1} \vee f_i \vee f_{i+1}))$,
- $\phi^s_3 = \Box((f_1 \wedge \bigcirc f_2) \vee (f_2 \wedge \bigcirc f_3) \rightarrow (b_1 \vee b_2 \vee b_3))$, and
- $\phi^s_{4,j} = \Box\Diamond(b_j \rightarrow f_j)$.

The requirement ϕ^s_3 says that the lift moves up one floor only if some button is pressed. The specification $\phi = \phi_e \rightarrow \phi_s$ is realizable. Now assume that the designer wants to ensure that all floors are infinitely often visited; thus she adds the guarantees $\bigwedge_j \phi^s_{5,j}$ where $\phi^s_{5,j} = \Box\Diamond(f_j)$ to the set of system requirements. The specification $\phi' = \phi_e \rightarrow \phi_s \bigwedge_j \phi^s_{5,j}$ is not realizable. A counter-strategy for the environment is to always keep all b_i's low. We run our algorithms with the set of all the environment variables $\{b_1, b_2, b_3\}$ for all assumption forms. The algorithm generates the refinements $\psi_1 = \Box\Diamond(b_1 \vee b_2 \vee b_3)$ and $\psi_2 = \Box((\neg b_1 \wedge \neg b_2 \wedge \neg b_3) \rightarrow \bigcirc(b_1 \vee b_2 \vee b_3))$. Refinement ψ_1 requires that the environment infinitely often presses a button. Refinement ψ_2 is another suggestion which requires the environment to make a request after any inactive turn. Refinement ψ_1 seems to be more reasonable and the user can add it to the specification to make it realizable.

Only one counter-strategy is processed during the search for finding refinements and three candidate assumptions are generated overall, where one of the candidates is inconsistent with ϕ' and the two others are refinements ψ_1 and ψ_2. Thus, the search terminates after checking the generated assumptions at first level. Only 0.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategy. Note that to generate ψ_1 using the template-based method in [7], the user needs to specify a template with three variables which leads to $2^3 = 8$ candidate assumptions, although only one of them is satisfied by the counter-strategy.

B. AMBA AHB

ARM's Advanced Microcontroller Bus Architecture (AMBA) defines the Advanced High-Performance Bus (AHB) which is an on-chip communication protocol. Up to 16 *masters* and 16 *slaves* can be connected to the bus. The masters start the communication (read or write) with a slave and the slave responds to the request. Multiple masters can request the bus at the same time, but the bus can only be accessed by one master at a time. A bus access can be a single *transfer* or a *burst*, which consists of multiple number of transfers. A bus access can be locked, which means it cannot be interrupted. Access to the bus is controlled by the *arbiter*. More details of the protocol can be found in [3]. We use the specification given by one of RATSY's example files (amba02.rat). There are four environment signals:

- $\texttt{HBUSREQ}[i]$: Master i requests access to the bus.
- $\texttt{HLOCK}[i]$: Master i requests a locked access to the bus. This signal is raised in combination with $\texttt{HBUSREQ}[i]$.
- $\texttt{HBURST}[1:0]$: Type of transfer. Can be SINGLE (a single transfer), BURST4 (a four-transfer), or INCR (unspecified length burst).
- $\texttt{HREADY}$: Raised if the slave has finished processing the data. The bus owner can change and transfers can start only when HREADY is high.

The first three signals are controlled by the masters and the last one is controlled by the slaves. The specification of amba02.rat consists of one master and two slaves. For our experiment, we remove the fairness assumption $\Box\Diamond\texttt{HREADY}$ from the specification. The new specification is unrealizable. We run our algorithm with the sets of variables $\{\texttt{HREADY}\}$, $\{\texttt{HREADY}, \texttt{HBUSREQ}[0], \texttt{HBUSREQ}[1], \texttt{HLOCK}[0], \texttt{HLOCK}[1]\}$, $\{\texttt{HREADY}\}$ and $\{\texttt{HBUSREQ}[0], \texttt{HBUSREQ}[1]\}$ to be used in liveness, safety, left and right hand side of transition assumptions, respectively. Some of the refinements generated by our method are: $\psi_1 = \Box\Diamond\texttt{HREADY}$, $\psi_2 = \Box(\texttt{HREADY} \vee \neg\texttt{HBUSREQ}[0] \vee \neg\texttt{HLOCK}[0] \vee \neg\texttt{HBUSREQ}[1] \vee \neg\texttt{HLOCK}[1]) \wedge \Box\Diamond\texttt{HREADY}$, and $\psi_3 = \Box(\texttt{HREADY} \rightarrow \bigcirc\neg\texttt{HBUSREQ}[0]) \wedge \Box(\neg\texttt{HREADY} \rightarrow \bigcirc\neg\texttt{HBUSREQ}[0])$. Note that although ψ_2 is a consistent refinement, it includes ψ_1 as a subformula and it is more restrictive. The refinement ψ_3 implies that $\texttt{HBUSREQ}[0]$ must always be low from the second step on. Among these suggested refinements, ψ_1 appears to be the best option. Our method only processes one counter-strategy with five states and generates five candidate assumptions to find the first refinement ψ_1. To find all refinements within the depth two, overall five counter-strategies are processed by our method during the search, where the largest counter-strategy had 25 states. The number of assumptions generated for each counter-strategy during the search is less than nine. 28.6 percent of total computation time was spent on generating candidate assumptions from the counter-strategies.

VI. Conclusion and Future Work

We presented a counter-strategy guided approach for adding environment assumptions to an unrealizable specifications in order to achieve realizability. We gave algorithms for synthesizing weakest assumptions of certain forms (based on "patterns") that can be used to rule out the counter-strategy.

We chose to apply explicit-state graph search algorithms on the counter-strategy because the available tools for solving games output the counter-strategy as a graph in an explicit form. Symbolic analysis of the counter-strategy may be desirable for scalability, but the key challenge for this is to develop algorithms for solving games that can produce counter-examples in compact symbolic form. Synthesizing symbolic patterns is one of the future directions.

Counter-strategies provide useful information for explaining reasons for unrealizability. However, there can be multiple ways to rule out a counter-strategy. We plan to investigate how the multiplicity of the candidates generated by our method can be used to synthesize better assumptions. Furthermore, our method asks the user for subsets of variables to be used in generating candidates. The choice of the subsets can significantly impact how fast the algorithm can find a refinement. Automatically finding good subsets of variables that contribute to the unrealizability problem is another future direction. Synthesizing environment assumptions for more general settings, and using the method for synthesizing interfaces between components in context of compositional synthesis are subject to our current work.

References

[1] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of gr(1) temporal logic specifications. Technical report. arXiv:1308.4113 [cs.LO].

[2] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy–a new requirements analysis tool with synthesis. In *CAV 2010*, pages 425–429. Springer, 2010.

[3] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[4] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *CONCUR 2008*, pages 147–161. Springer, 2008.

[5] R. Konighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD 2009*, pages 152–159, 2009.

[6] O. Kupferman, N. Piterman, and M. Vardi. Safraless compositional synthesis. In *CAV 2006*, pages 31–44. Springer, 2006.

[7] W. Li, L. Dworkin, and S. Seshia. Mining assumptions for synthesis. In *MEMOCODE 2011*, pages 43–50. IEEE, 2011.

[8] K. McMillan. Cadence SMV. http://www.kenmcmil.com/smv.html.

[9] N. Ozay, U. Topcu, and R. Murray. Distributed power allocation for vehicle management systems. In *CDC-ECC 2011*, pages 4841–4848. IEEE, 2011.

[10] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In *VMCAI 2006*, pages 364–380. Springer, 2006.

[11] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL 1989*, pages 179–190. ACM, 1989.

[12] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.